EPITA

Rapport de Projet

Projet J3D

Reach of Vespers

Studio J-1

Membres de l'équipe :
Jules Hoeltzel
Thomas Machado–Da Assuncao
Thibault Ansart
Clément Genty
Tristan Lempereur

Année 2025 - 2026

Table des matières

1 Introduction **2**
- 1.1 Présentation du projet et ambitions initiales . . . 2
- 1.2 Présentation de l'équipe et rôle de chacun . . . 2

2 Reprise du Cahier des Spécifications Techniques (TSD) **3**
- 2.1 Respect des contraintes pédagogiques . . . 3
- 2.2 Identité du jeu et mélange des genres . . . 3
- 2.3 Production des ressources et écosystème . . . 3
- 2.4 Bilan prévisionnel et justification des écarts . . . 4

3 Développement et répartition du travail **5**
- 3.1 Une répartition dimensionnée et basée sur les compétences . . . 5
- 3.2 Modalités de communication et de synchronisation . . . 5
- 3.3 Architecture globale et intégration . . . 6
- 3.4 Développements individuels . . . 6
  - 3.4.1 Jules Hoeltzel : Intelligence Artificielle, Interface, Collisions et Compilation . . . 6
  - 3.4.2 Thomas Machado–Da Assuncao : Multijoueur et Game Design . . . 14
  - 3.4.3 Thibault Ansart : Mini-jeux et Site Web . . . 19
  - 3.4.4 Clément Genty : Monstres/Boss, Classe Joueur, Tutoriel, Système de sauvegarde et Marketing . . . 27
  - 3.4.5 Tristan Lempereur : Inventaire, Shop et Sound Design . . . 36

4 Récit de la Réalisation **46**
- 4.1 Nos réussites et nos joies . . . 46
- 4.2 Nos peines et les défis rencontrés . . . 46
- 4.3 Analyse critique et solutions apportées . . . 47

5 Qualité du Produit et Documentation Technique **48**
- 5.1 Direction artistique et prise en main . . . 48
- 5.2 Un exécutable unique et robuste . . . 48
- 5.3 Déploiement et Dossier d'Exploitation . . . 48
- 5.4 Rigueur de la documentation technique . . . 49

6 Conclusion Générale **50**

7 Annexe **52**
- 7.1 Le HUD et le menu Pause . . . 52
- 7.2 Les items . . . 53

Chapitre 1

Introduction

1.1 Présentation du projet et ambitions initiales

Dans le calme de sa chambre, Soren, un samouraï d'une rigueur implacable, découvre un artefact singulier sur son bureau : un ouvrage à la couverture vierge et aux pages désespérément blanches. Poussé par une curiosité inhabituelle, il l'ouvre. En un battement de cils, la réalité se déchire. Soren est absorbé par le livre et projeté dans une dimension lugubre, rongée par une magie cauchemardesque.

Reach of Vespers est un jeu de plateforme en deux dimensions inspiré du style Metroidvania et de références majeures telles que Hollow Knight. Développé entièrement en Python en s'appuyant sur le framework graphique, PyGame, il propose un univers unique et des phases de gameplay diversifiées mêlant exploration, combats, mini-jeux et collecte d'objets.

1.2 Présentation de l'équipe et rôle de chacun

Notre équipe, le Studio J-1, se compose de cinq étudiants de l'EPITA animés par un objectif commun : produire un projet informatique qualitatif, esthétique et immersif, tout en proposant une grande richesse de mécaniques de jeu.

- **Jules Hoeltzel** : Coordinateur du projet, développement des IA, des collisions, de l'interface utilisateur et des cinématiques.
- **Thomas Machado–Da Assuncao** : Game Designer, développement et organisation des niveaux, ainsi que le multijoueur. Développement des objets à collectionner.
- **Thibault Ansart** : Développeur Web et mini jeux.
- **Clément Genty** : Développeur Commercial, développement des monstres et support manette.
- **Tristan Lempereur** : Sound Designer et développement du système d'inventaire.

Chapitre 2

Reprise du Cahier des Spécifications Techniques (TSD)

2.1 Respect des contraintes pédagogiques

Conformément aux contraintes imposées par l'école, notre jeu est développé en langage Python en s'appuyant sur le framework graphique, PyGame. L'ensemble des contraintes techniques ont été respectées et intégrées de la manière suivante :

- **Intelligence Artificielle :** Comme stipulé dans notre cahier des spécifications techniques, la totalité des ennemis sont dotés de comportements autonomes.
- **Multijoueur en réseau :** Le mode coopératif imposé s'organise à travers le réseau local (LAN).

2.2 Identité du jeu et mélange des genres

Afin de nous démarquer, nous avons fait le choix de fusionner plusieurs styles vidéoludiques :

- Le cœur du jeu repose sur les mécaniques du Platformer et de l'Action-Adventure.
- Nous y avons intégré des fonctionnalités tirées du RPG, notamment via le système d'évolution du personnage et de loot (récupération d'objets).
- Les affrontements dynamiques s'inspirent des jeux de Fighting et Beat them all.
- Enfin, les mini-jeux offrent une dimension Puzzles permettant de diversifier le gameplay.

2.3 Production des ressources et écosystème

Dans la production de nos ressources (assets), nous avons respecté l'univers sombre et mystérieux de notre direction artistique initiale :

- **Graphismes :** Mêlant créations personnalisées et ressources existantes, le personnage principal, Soren, a été modélisé à partir de croquis originaux.
- **Audio :** Les musiques et bruitages ont été rigoureusement sélectionnés sur la plateforme Pixabay, qui recense des sons libres de droits.
- **Écosystème Web :** Comme convenu, le site web vitrine a été conçu intégralement sur mesure par Thibault, afin d'offrir une expérience optimale et d'inciter au téléchargement du jeu.

2.4 Bilan prévisionnel et justification des écarts

Notre bilan prévisionnel visait un achèvement des tâches à 100% pour la soutenance finale.

Cependant, le cycle de développement a nécessité d'importantes adaptations techniques et organisationnelles par rapport à nos prévisions initiales :

- **Réorganisation face aux retards :** Lors de la deuxième soutenance, notre avancement sur le multijoueur n'était que de 30% et nous accusions un retard sur la richesse du contenu.
- **Évolution du Framework :** Nous avons été techniquement limités par la version officielle de PyGame, qui ne proposait pas d'accélération matérielle.
- **Solutions d'affichage vidéo :** Pygame ne proposant pas de lecteur vidéo natif pour nos cinématiques, nous avons dû chercher une solution alternative et intégrer la librairie OpenCV, qui a permis la lecture fluide de nos vidéos au sein même du jeu.

Chapitre 3

Développement et répartition du travail

3.1 Une répartition dimensionnée et basée sur les compétences

Afin de garantir l'investissement et la motivation de chaque membre, nous avons procédé par sondages pour répartir les tâches en fonction des affinités de chacun. Notre objectif était de respecter les souhaits individuels tout en agissant systématiquement dans l'intérêt du groupe.

Nous avons particulièrement veillé à dimensionner la charge de travail en fonction de la complexité technique des missions. Par exemple, si l'on observe un écart dans le nombre de tâches principales attribuées (Clément étant responsable de 4 missions contre 2 pour Thomas), cela résulte d'un choix stratégique : le développement du mode multijoueur étant l'un des défis techniques les plus ardus du projet, nous avons volontairement allégé les autres responsabilités de Thomas pour lui permettre de s'y consacrer pleinement. De plus, les missions ont été ciblées selon les acquis personnels de chacun ; les connaissances préalables de Clément en marketing ont par exemple fait de lui le candidat idéal pour ce pôle.

Pour sécuriser cette répartition, nous nous sommes appuyés sur le système de "Responsable (R) / Suppléant (S)" défini dans notre cahier des spécifications techniques. Cette méthode a favorisé une entraide constante : chaque responsable savait qu'il pouvait s'appuyer sur son suppléant en cas de blocage technique.

3.2 Modalités de communication et de synchronisation

Pour assurer le suivi du projet, nous avons justifié nos choix d'outils en fonction de nos objectifs et de nos méthodes de travail. Notre principal canal de communication a été un serveur Discord dédié, qui nous a permis de centraliser nos échanges, de partager nos ressources (assets) et d'organiser des réunions vocales régulières. Ces points de synchronisation à distance ont été complétés par des réunions en présentiel pour les prises de décisions majeures ou les sessions d'entraide complexes.

Pour le développement, nous avons utilisé GitHub afin de versionner notre code. Cet outil, dont nous avons pu maîtriser l'utilisation grâce aux cours dispensés à l'EPITA, nous a garanti une mise en commun rapide et sécurisée de nos différentes branches.

Enfin, face aux divers retards rencontrés en cours d'année, notre infrastructure de communication nous a permis d'être agiles : nous avons pu organiser des réunions d'urgence

sur Discord pour réorganiser nos plannings de manière collégiale, en veillant à prendre des décisions adaptées qui n'entravent pas la dynamique et la motivation de l'équipe.

3.3 Architecture globale et intégration

La force de notre produit final réside dans la parfaite intégration des modules développés individuellement. La boucle principale de notre programme fait appel à la totalité de ces parties (à l'exception du site web vitrine) afin d'harmoniser le jeu et de créer une expérience utilisateur fluide et unique.

Concrètement, lors du lancement, l'interface utilisateur et les cinématiques conçues par Jules offrent un premier rendu visuel immersif. Le programme intègre ensuite les systèmes de sauvegarde et de gestion de données développés par Clément, tout en proposant l'accès au mode multijoueur structuré par Thomas. Une fois la partie chargée, l'intelligence artificielle, l'environnement sonore de Tristan et les mécaniques de jeu fusionnent en un seul exécutable pour offrir au joueur l'expérience Reach of Vespers dans son intégralité.

3.4 Développements individuels

3.4.1 Jules Hoeltzel : Intelligence Artificielle, Interface, Collisions et Compilation

En tant que coordinateur technique de Reach of Vespers, ma responsabilité première a été d'architecturer et de développer les fondations interactives du moteur de jeu. Mon périmètre d'action s'est concentré sur la conception de l'intelligence artificielle des entités, la programmation de l'interface utilisateur (menus, HUD et expérience utilisateur), ainsi que l'implémentation du moteur physique gérant les collisions. Au-delà de ces piliers, j'ai adopté une posture de support (en tant que suppléant) sur l'architecture multijoueur et le développement des mini-jeux afin d'épauler l'équipe dans les moments critiques, avant de prendre en charge l'entièreté du processus de compilation et de déploiement du produit final.

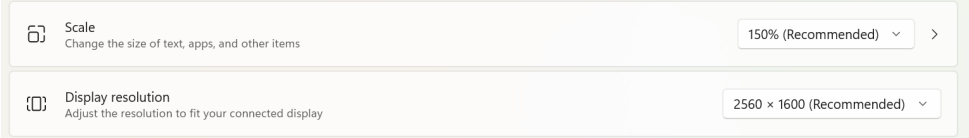
L'interface Utilisateur (UI) et le Moteur de Rendu

La genèse de notre développement visuel a reposé sur la création d'un moteur "scaler" qui ingère des valeurs dimensionnelles pensées pour une résolution de référence (codées en dur) et restitue des coordonnées absolues adaptées à l'écran de l'utilisateur, en utilisant un calcul croisé basé sur des pourcentages de la largeur (width) et de la hauteur (height) de la fenêtre active.

```
class UIScaler:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def scale(self, px=0, py=0, pw=0, ph=0):
        return (int(self.width * px), int(self.height * py),
                int(self.width * pw), int(self.height * ph))
```

Cependant, j'ai rapidement été confronté à une problématique système particulièrement tenace : la mise à l'échelle (DPI scaling) native de Windows. Sur Windows 10 et 11, il est courant que les utilisateurs appliquent un zoom système (par exemple 125% ou 150%) pour le confort de lecture.



Ce paramètre faussait totalement nos calculs : les requêtes demandant la résolution de l'écran renvoyaient des valeurs altérées par ce ratio, détruisant ainsi l'affichage du jeu. Pour contourner ce comportement, j'ai dû descendre à un niveau plus bas et interagir directement avec l'environnement de l'OS via la bibliothèque Python native `os`. J'ai pu forcer le processus du jeu à s'annoncer comme DPI-aware (ignorant la mise à l'échelle de Windows et forçant le ratio à 100% en interne). C'est cette manœuvre technique qui a garanti l'universalité de notre affichage sur l'ensemble des configurations matérielles.

Une fois la base de rendu stabilisée, je me suis attaqué à la logique de navigation. Un jeu vidéo n'étant techniquement qu'une boucle infinie (la Game Loop), il fallait concevoir une architecture logicielle capable de changer radicalement le contexte d'affichage et de logique sans briser cette boucle. J'ai donc implémenté le patron de conception State Machine (Machine à états). En exploitant la programmation orientée objet, j'ai doté notre gestionnaire principal d'un attribut `state`. À chaque itération de la boucle, le programme évalue cet état (ex : MENU, PLAYING, SETTINGS) et délègue l'affichage et la gestion des événements à la classe correspondante. Cette architecture modulaire a rendu la création de sous-menus extrêmement fluide et maintenable.

Immersion et intégration vidéo

Pour accroître l'immersion dès les premières secondes, j'ai souhaité intégrer une cinématique d'introduction présentant le titre du jeu et le logo de notre studio. Pygame ne supportant pas nativement le décodage vidéo, j'ai dû intégrer la librairie OpenCV. J'ai conçu un lecteur vidéo personnalisé qui extrait la vidéo frame par frame, redimensionne chaque image à la volée via notre scaler, et les affiche sur une surface Pygame en synchronisant le tout sur le framerate (FPS) du jeu. Pour maintenir un code propre (Clean Code), cette logique a été encapsulée dans une classe dédiée, permettant de lancer une cinématique depuis n'importe où via un simple appel de méthode explicite tel que `video.play()`.

Paramétrages, Accessibilité et Expérience Utilisateur

Le menu des paramètres a été itéré tout au long du cycle de vie du projet. Il offre aujourd'hui un contrôle granulaire à l'utilisateur : bascule du mode plein écran, gestion des préréglages graphiques ("High"/"Low" influençant le niveau de détail), et un mixeur audio complet permettant de dissocier les effets sonores (SFX) des musiques d'ambiance.

Concernant les contrôles, j'ai mis en place un système de Key Binding dynamique pour les joueurs clavier. Ils peuvent remapper chaque action, les nouvelles saisies étant interceptées, validées, puis sérialisées et stockées dans un fichier de configuration local (sauvegarde persistante entre deux lancements). Un bouton de réinitialisation ("Default") permet de restaurer les paramètres d'usine en cas d'erreur.

Pour le jeu à la manette, j'ai adopté une approche différente. Après une phase d'étude ergonomique sur des jeux de plateforme et d'action contemporains, j'ai défini un mapping universel optimisé (non modifiable pour garantir l'expérience de jeu). Lorsque le moteur détecte un contrôleur (via le module joystick), l'interface s'adapte instantanément, affichant une cartographie complète et visuelle de la manette, avec une détection automatique du type de périphérique (comme l'explicitera Clément dans sa partie).

Nous avions initialement prototypé un menu sous forme de "Hub jouable", où le personnage devait se déplacer physiquement pour lancer la partie ou changer les options. Si l'idée était séduisante sur le papier, les tests utilisateurs ont révélé une friction trop importante : les contrôles n'étaient pas encore paramétrables dynamiquement dans ce hub, et cela imposait une phase de gameplay non pertinente avant même de lancer l'aventure. Nous avons donc pris la décision architecturale de revenir à une interface utilisateur traditionnelle, beaucoup plus ergonomique, directe et efficace.

Enfin, pour lier le menu à l'expérience de jeu, j'ai conçu un écran de sélection de sauvegarde interactif. Une carte globale du monde de Reach of Vespers s'affiche, et le système lit les métadonnées de progression pour débloquer les différents points d'apparition, permettant au joueur de reprendre son aventure exactement où il le souhaite. Ce travail a nécessité une synchronisation étroite avec Clément et Thomas pour lier l'UI aux systèmes de sauvegarde, de multijoueur et d'entrées manette.

L'Affichage Tête Haute (HUD) et le Menu Pause

En jeu, j'ai opté pour un HUD (Head-Up Display) diégétique et minimaliste afin de maximiser la lisibilité de l'action. Il comprend :

- Une barre de vie dynamique sous forme de cœurs. L'algorithme gère l'augmentation de la capacité maximale (gain d'endurance) et grise les cœurs lors de la prise de dégâts, avec un retour visuel immédiat.
- Un compteur de monnaie mis à jour en temps réel.
- Un système de notification de zone (pop-up discret en haut à droite) informant le joueur de son emplacement.
- Un affichage contextuel d'interaction : lorsqu'une collision est détectée avec une entité ou un objet manipulable, un indicateur flottant "Interact" apparaît au-dessus de l'élément, guidant le joueur sans surcharger l'écran.

Les ennemis communs disposent de jauges de santé locales ancrées au-dessus de leurs sprites, tandis que les Boss bénéficient d'une barre de vie magistrale en haut de l'écran, soulignant l'importance de l'affrontement.

Le système de pause a demandé une attention particulière. En mode solo, l'activation du menu met en pause l'horloge interne (le delta_time) et stoppe le rafraîchissement des entités, figeant véritablement le monde. En revanche, pour garantir la stabilité de l'architecture réseau en multijoueur, j'ai dû implémenter une exception conditionnelle : l'ouverture du menu affiche bien l'interface (permettant de sauvegarder ou de régler les options), mais le monde continue d'être mis à jour en arrière-plan pour ne pas désynchroniser l'expérience de l'autre joueur.

(cf. Annexe pour voir les visuels.)

L'Intelligence Artificielle (IA)

Pour donner vie à l'univers de Reach of Vespers et offrir un défi constant (facile à prendre en main, difficile à maîtriser), j'ai programmé les comportements algorithmiques de nos entités. Le système repose sur des Machines à États Finis (FSM) dictant les routines de "Wander" (déambulation) et de "Pathfinding" (recherche de cible) des ennemis, développées entièrement en Python pur sans bibliothèques externes.

Comportement standard et Edge Detection : À l'état neutre, un monstre patrouille aléatoirement sur sa plateforme. Le défi mathématique ici n'était pas le mouvement, mais la perception de l'environnement : comment empêcher un ennemi basique de tomber d'une corniche ? J'ai implémenté un système de "raycasting virtuel" simplifié. Lors de l'initialisation du monstre, des boîtes de collisions invisibles (Rects) sont générées et projetées dynamiquement aux coins inférieurs de sa Hitbox (en fonction de la direction de son regard). À chaque frame, l'algorithme vérifie si cette boîte virtuelle entre en collision avec la couche du sol. Si la réponse est nulle (la boîte "flotte" dans le vide), le monstre comprend qu'il a atteint un bord et inverse instantanément son vecteur de vélocité pour faire demi-tour. Ce calcul est hautement optimisé car il ne vérifie qu'un seul point d'impact anticipé.

```
def check_edge(self, platforms):
    detect_dist = 20
    box_x = self.hitbox.right if self.facing_direction == 1 else
       self.hitbox.left
    self.detect_box = pygame.Rect(box_x, self.hitbox.bottom,
       detect_dist, 10)

    for platform in platforms:
        if self.detect_box.colliderect(platform):
            return False

    return True
```

Système d'Aggro et Ciblage Multijoueur : Dès qu'un joueur entre dans le rayon de détection d'un ennemi, l'IA bascule en état de "Chasse" : elle cible le joueur, adapte sa vitesse et initie une poursuite. Le monstre est capable de résoudre ses animations d'attaque tout en conservant son inertie de déplacement, rendant le kiting (fuir tout en attaquant) beaucoup plus technique pour le joueur. De plus, une fois alertée, le rayon de "désaggro" (perte de vue) de l'IA est multiplié, la rendant tenace.

L'intégration du multijoueur a considérablement complexifié cette logique. L'IA devait désormais évaluer et choisir entre deux cibles potentielles. J'ai développé un algorithme de priorisation : le monstre calcule la distance avec les deux joueurs et verrouille le plus proche. Pour éviter un comportement erratique où le monstre hésiterait constamment entre les deux cibles, la cible est verrouillée tant qu'elle reste à portée. Cependant, j'ai intégré une notion "d'interruption d'aggro" : si le joueur B attaque le monstre pendant qu'il pourchasse le joueur A, le monstre met à jour son contexte et se retourne vers son nouvel agresseur. Cette mécanique systémique permet aux joueurs de véritablement coopérer, l'un pouvant "tanker" ou attirer l'attention (kite) pendant que l'autre inflige des dégâts.

Les Boss bénéficient d'un arbre décisionnel plus poussé. En plus de ces mécaniques, ils évaluent en temps réel la distance les séparant du joueur pour choisir leur pattern d'attaque de manière adaptative : déclenchement de scripts d'attaques au corps-à-corps si

la cible est proche, ou calcul de trajectoire pour tirer des projectiles orientés si le joueur tente de rester à distance.

Le Moteur Physique et les Collisions

Le cœur du gameplay réside dans son Game Feel, lui-même directement lié à la précision de la physique. Dans un premier temps, j'ai mis en place un parseur lisant les métadonnées de notre Tilemap (exportée depuis le logiciel Tiled). Ce script génère dynamiquement une couche d'objets invisibles correspondant à la topologie du terrain. À chaque itération de la physique, le moteur anticipe la future position du joueur en fonction de ses vecteurs de vitesse (X et Y). S'il détecte une intersection entre le rectangle (hitbox) du joueur et un élément de la carte, le déplacement sur l'axe concerné est instantanément annulé, empêchant de traverser les murs.

Mécaniques avancées de déplacement : J'ai utilisé ce système pour créer des mécaniques de plateforme avancées. Si l'algorithme détecte une collision continue sur l'axe X (le joueur pousse contre un mur) alors qu'il est en l'air (vélocité Y positive), j'applique une friction artificielle qui réduit la vitesse de chute, et je déclenche l'animation de glissade (Wall-Slide). Depuis cet état, le compteur de sauts est réinitialisé, autorisant le joueur à exécuter un Wall-Jump en s'appuyant sur le mur. Pour éviter les comportements abusifs (exploits), j'ai rigoureusement codé des variables d'états booléennes (`is_grounded`, `can_jump`) qui s'assurent que la gravité s'applique de manière réaliste et qu'un saut en l'air n'est autorisé que dans des contextes stricts (double saut débloqué, ou saut mural).

La Couche Interactive

Au-delà des collisions, j'ai élaboré une "couche de collisions interactives". Il s'agit de zones invisibles qui n'entravent pas le mouvement, mais qui, lors d'une intersection avec

le joueur, se mettent en écoute des entrées clavier/manette. Si la touche d'interaction est pressée, la fonction `trigger()` de l'objet est appelée (ouverture de porte, dialogue, coffre). J'ai travaillé en étroite collaboration avec Thomas (notre Game Designer) pour créer un pont technologique entre son outil de création (Tiled) et mon code. Il lui suffit désormais d'ajouter des propriétés personnalisées (Custom Properties) à ses tuiles dans Tiled pour que mon parseur reconnaisse automatiquement l'objet, lui applique la bonne boîte de collision interactive, et relie le bon script d'événement, le tout sans qu'il n'ait jamais à toucher à une seule ligne de code Python.

L'Ingénierie de Déploiement : Compilation et Distribution

La qualité d'un logiciel se juge aussi à sa facilité de distribution. Fournir un code source Python et demander aux testeurs d'installer des dépendances n'était pas envisageable pour un produit que l'on souhaitait fini et professionnel. J'ai donc pris en charge la chaîne de compilation logicielle. J'ai exploité la librairie `pyinstaller`. La complexité ne résidait pas dans la compilation en elle-même, mais dans la gestion du fichier de spécification (`.spec`). Notre projet possédant une arborescence complexe (assets graphiques, fichiers audio, polices, shaders OpenCV, librairies dynamiques externes), j'ai dû scripter ce fichier `.spec` pour qu'il emballe (bundle) l'intégralité de ces ressources dans un exécutable unique (Standalone), tout en gérant les "hidden imports" (dépendances requises à l'exécution mais non détectées statiquement par le compilateur).

Pour aller au bout de la démarche, j'ai transformé cet exécutable brut en une véritable expérience d'installation via l'outil de création de scripts Inno Setup. J'ai programmé un "Wizard Installer" (Assistant d'installation) en anglais, respectant scrupuleusement les standards de l'industrie (UX d'installation classique : acceptation, choix du répertoire cible, création optionnelle d'un raccourci bureau). Le script Inno Setup prend également en charge la création de clés de registre Windows. Cela permet à Reach of Vespers d'apparaître officiellement dans la liste des programmes et fonctionnalités de l'OS. En cas de suppression, l'exécutable de désinstallation natif (Uninstall Wizard) s'assure d'un nettoyage parfait du système, purgeant les fichiers du jeu et les dépendances sans laisser de fichiers orphelins sur la machine de l'utilisateur.

Vision Transversale et Contributions d'Équipe

Dans la gestion d'un projet de cette envergure, le rôle de coordinateur implique de l'agilité et un décloisonnement des tâches. Pour maintenir le momentum du développement et éviter que le chemin critique du projet ne soit bloqué, j'ai fréquemment opéré en dehors de mon périmètre strict. Par exemple, le développement de l'intelligence artificielle exigeait des entités sur lesquelles la tester. L'équipe en charge de l'intégration des monstres ayant pris du retard, j'ai pro-activement développé le Minimum Viable Product (MVP) de la classe Enemy. J'ai structuré la hiérarchie de l'objet, son intégration dans le groupe de sprites Pygame et ses variables de base, ce qui m'a permis d'avancer sur mon IA de manière asynchrone, sans attendre la livraison finale des assets et de la logique de dégâts.

De la même manière, j'ai pris l'initiative d'architecturer les ponts techniques entre mon menu principal et l'instanciation de la boucle de jeu de mes collaborateurs. Connaissant intimement les variables environnementales et les structures de données générées par mon menu (sélection de difficulté, choix du mode multijoueur, configuration des touches), j'ai pu injecter ces paramètres proprement dans le constructeur du jeu principal, facilitant

grandement la tâche du reste de l'équipe et garantissant une transition fluide et sans fuite de mémoire entre l'interface logicielle et l'expérience ludique.

3.4.2 Thomas Machado–Da Assuncao : Multijoueur et Game Design

Le Multijoueur

Le développement du mode multijoueur de Reach of Vespers représente l'un des aspects techniques les plus importants et les plus compliqués du projet. Dès le début du développement, nous avons souhaité proposer une expérience coopérative permettant à deux joueurs d'explorer ensemble le monde du jeu, d'affronter les ennemis en coopération et de progresser simultanément dans les différentes zones de l'aventure. L'objectif principal était de garantir une expérience fluide dans laquelle les deux joueurs évoluent dans le même monde sans ressentir de latence importante ni observer d'incohérences entre les deux instances du jeu.

Le système multijoueur repose sur une architecture client-serveur. Dans notre cas, l'un des deux joueurs héberge la partie et joue le rôle de serveur tandis que le second rejoint la partie en tant que client via le réseau local. Le serveur devient alors responsable de tous les éléments liés au gameplay. Il gère les déplacements des joueurs, les collisions, les dégâts, les interactions avec l'environnement, les ennemis, les récompenses ainsi que les changements de zone. Les clients, quant à eux, n'envoient pas directement l'état complet de leur personnage mais transmettent principalement leurs inputs au serveur, qui interprète ensuite ces actions dans le contexte du monde actuel avant de mettre à jour la simulation du jeu.

Le système de connexion du multijoueur de Reach of Vespers repose sur un mécanisme de code généré à partir de l'adresse IP locale du joueur qui héberge la partie. Lorsqu'un joueur choisit l'option Host, son ordinateur ouvre une session réseau locale et récupère automatiquement son adresse IP. Cette adresse IP est ensuite convertie en une valeur plus courte grâce à un encodage en base 36. Cela permet d'obtenir un code beaucoup plus simple à transmettre à l'autre joueur. L'adresse IP du host est d'abord transformée en un grand nombre entier. Ce nombre est ensuite converti en base 36, c'est-à-dire avec les chiffres de 0 à 9 et les lettres de A à Z. Le résultat est un code court et lisible que le joueur peut communiquer au second joueur. De son côté, le joueur qui souhaite rejoindre la partie entre ce code dans le menu Join. Le jeu effectue alors l'opération inverse : il reconvertit le code base 36 en adresse IP, puis utilise cette adresse pour se connecter au host.

La communication entre les deux instances repose sur des sockets TCP et sur des messages structurés au format JSON. Chaque message possède un type spécifique permettant d'identifier sa fonction. Certains messages servent à gérer la connexion des joueurs tandis que d'autres permettent de transmettre les inputs ou les états du monde. Les données échangées sont converties en JSON avant d'être envoyées via le réseau. Afin de gérer correctement les flux réseau, un système de buffer de réception a été mis en place. Les messages sont séparés par des retours à la ligne puis reconstruits progressivement afin de reformer les objets JSON complets. Le serveur conserve en permanence une représentation de l'état du monde et envoie régulièrement des snapshots aux clients. Ces snapshots contiennent les informations nécessaires à l'affichage du jeu, notamment les positions des entités, leurs animations ou certaines données liées au gameplay. À chaque frame, le joueur local prépare un dictionnaire contenant les informations importantes à synchroniser : position, animation, direction du personnage, état du joueur ou encore certaines interactions avec le monde.

L'une des principales difficultés rencontrées durant le développement du multijoueur concernait la synchronisation des différentes mécaniques du jeu. Dans les premières versions, de nombreux problèmes apparaissaient régulièrement. Il arrivait par exemple que les deux joueurs ne voient pas exactement la même chose à l'écran : certains ennemis pouvaient être morts sur une instance mais encore vivants sur l'autre, les positions des personnages pouvaient être légèrement différentes et certains objets pouvaient apparaître de manière incohérente. Une grande partie du travail a donc consisté à corriger progressivement ces désynchronisations afin d'obtenir une expérience coopérative stable et cohérente. Même en réseau local, une légère latence peut apparaître. Afin de limiter cet effet, un système d'interpolation a été utilisé pour les entités distantes. Au lieu de mettre à jour brutalement les positions lors de la réception des snapshots, les mouvements sont affichés de manière progressive.

Les animations ont également nécessité plusieurs ajustements techniques. Certaines animations, notamment les attaques, étaient parfois interrompues trop rapidement lors de la réception de nouveaux états réseau. Afin de résoudre ce problème, un système de timer a été ajouté pour forcer l'affichage complet des animations importantes avant leur remplacement par un nouvel état. Le système réseau a ensuite été progressivement étendu à des mécaniques beaucoup plus complexes. Une grande partie du travail a notamment porté sur la synchronisation des drops d'objets, des récompenses, des boss, des armures ainsi que des fragments d'épée. Lorsqu'un ennemi est vaincu, les deux joueurs doivent voir apparaître exactement les mêmes objets au sol. Ces objets doivent ensuite pouvoir être récupérés correctement sans provoquer de duplication ni de disparition incohérente. Pour cela, les objets au sol sont envoyés aux clients sous forme de données réseau contenant leur position, leur type ou encore leur état physique.

Lorsqu'un boss est vaincu, les récompenses doivent être distribuées correctement aux deux joueurs. De la même manière, lorsqu'un joueur récupère un objet, l'information doit être envoyée au serveur afin d'éviter toute duplication ou désynchronisation des inventaires. Le fonctionnement des ennemis suit également cette logique. Le serveur reste responsable de leur comportement et de leur état. Il synchronise leur position, leur animation, leur direction et leurs points de vie auprès des clients. Lorsqu'un joueur client touche un ennemi, les dégâts ne sont pas appliqués directement localement. Une demande de hit est envoyée au serveur, qui vérifie ensuite l'ennemi concerné et lui applique les dégâts.

Les Items (cf. Annexe pour voir les visuels.)

Une très grande partie de mon travail a également porté sur les systèmes d'objets, d'équipements et de la connexion de ceux-ci avec l'inventaire du jeu. Après la mise en place des bases du multijoueur, j'ai développé l'ensemble des mécaniques liées aux items présents dans Reach of Vespers. L'objectif principal était d'ajouter une véritable boucle de progression au gameplay afin de rendre les combats et l'exploration plus récompensant. Au départ, le joueur ne pouvait ni récupérer ni utiliser d'objets. Il était donc nécessaire de créer un système complet permettant aux ennemis de faire apparaître des récompenses directement dans le monde du jeu tout en restant compatibles avec les autres systèmes déjà présents, notamment le multijoueur et les sauvegardes.

Le fonctionnement des drops repose sur la création d'objets après la mort d'un ennemi. Lorsqu'un ennemi est vaincu, un item est généré avec différentes propriétés physiques : position, vitesse verticale, collisions, état au sol et animation de flottement. L'objectif était de donner aux joueurs une récompense visuelle dans le monde afin de rendre leur récupération plus satisfaisante pour le joueur. Les objets apparaissent donc avec une animation de chute puis effectuent un léger mouvement vertical une fois posés au sol.

Une fois l'objet généré, celui-ci doit interagir correctement avec les collisions de la map afin d'éviter qu'il traverse le décor ou reste bloqué dans certaines plateformes. Une phase de mise à jour physique est donc appliquée à chaque frame afin de gérer la gravité et les collisions avec le sol. Après l'atterrissage de l'objet, un système de flottement est utilisé afin de rendre le drop plus visible visuellement. Cette animation est calculée grâce à une fonction sinusoïdale appliquée verticalement à l'objet.

Plusieurs types d'objets ont ensuite été intégrés au jeu, notamment les améthystes, les gemmes de soin ainsi que différentes gemmes d'amélioration temporaire comme les gemmes de dégâts, de vitesse, de saut ou de dash. Chaque objet possède sa propre image, son nom ainsi qu'une description affichée dans l'inventaire.

L'un des problèmes techniques importants rencontrés concernait justement la gestion des images des objets. Certains assets utilisés dans le jeu étaient beaucoup plus grands que les cases de l'inventaire. Dans les premières versions, certaines images dépassaient des emplacements prévus ou devenaient floues après redimensionnement. Afin de résoudre ce problème, un système de redimensionnement a été ajouté afin d'adapter automatiquement les images selon leur contexte d'affichage : coffre, sol ou inventaire.

L'objectif de l'inventaire était de créer un système interactif capable de stocker les objets du joueur, d'afficher leurs informations et de permettre leur utilisation directement depuis l'interface. L'inventaire repose sur une organisation en grille facilitant la navigation aussi bien au clavier qu'à la manette. Chaque case de l'inventaire contient :

— une image ;
— un nom ;
— une quantité ;
— une description ;
— un type d'objet associé.

Lorsqu'un joueur sélectionne un objet, plusieurs actions deviennent possibles selon sa catégorie : utilisation d'un consommable, équipement d'une armure ou affichage des informations détaillées. L'une des difficultés importantes rencontrées concernait la compatibilité manette. Au départ, l'inventaire fonctionnait uniquement au clavier. J'ai donc développé un système de navigation permettant de déplacer la sélection grâce au joystick ou à la croix directionnelle. Cela nécessitait une gestion spécifique des inputs afin d'éviter des déplacements trop rapides ou incohérents dans la grille.

Les armures représentent également une partie importante du système d'équipement. Chaque boss du jeu peut désormais faire tomber une armure spécifique liée à sa région. Ces équipements permettent de réduire les dégâts reçus par le joueur et renforcent le sentiment de progression tout au long de l'aventure. Les principales armures du jeu sont Crimson Warden, Wraithguard Armor, Voidborn Shell et Crystal Bloom Plate.

Le système d'équipement permet au joueur d'activer une armure directement depuis l'inventaire. Lorsqu'une armure est équipée, ses effets sont appliqués immédiatement au personnage. Afin d'améliorer la lisibilité de l'interface, l'équipement actuellement utilisé est mis en évidence grâce à un contour spécifique.

Le système d'armes a lui aussi fortement évolué au cours du développement. Initialement, plusieurs épées différentes devaient être récupérées dans les différentes zones du jeu. Finalement, j'ai proposé de remplacer cette idée par un système plus cohérent avec l'univers du projet : les fragments de l'arme légendaire Vespers Edge.

Le joueur récupère progressivement plusieurs fragments répartis dans les différentes régions du monde. Chaque fragment augmente les dégâts du joueur jusqu'à la reconstruction complète de l'arme finale. Cette mécanique permet de relier directement la progression du gameplay à l'exploration du monde. Chaque région importante du jeu possède son propre fragment.

L'une des plus grandes difficultés de cette partie du projet concernait la compatibilité entre tous les systèmes. Les drops, l'inventaire, les armures, les fragments d'épée, les sauvegardes et le multijoueur devaient tous fonctionner ensemble sans provoquer de conflits. Plusieurs bugs apparaissaient régulièrement comme des objets dupliqués, équipements perdus après sauvegarde, récompenses invisibles ou incohérences entre les deux joueurs en multijoueur. Afin de résoudre ces problèmes, le but a été de centraliser les informations importantes du joueur et à sécuriser les interactions entre les différents systèmes. Les objets récupérés, les armures équipées et les fragments obtenus sont désormais correctement synchronisés et sauvegardés afin de garantir une progression stable et cohérente tout au long de l'aventure. Finalement, ces systèmes représentent aujourd'hui une partie essentielle du gameplay de Reach of Vespers. Ils permettent non seulement de renforcer la progression du joueur mais aussi d'ajouter davantage de profondeur à l'exploration, aux combats et à l'expérience globale du jeu.

Game Design des niveaux

La conception des niveaux (Level Design) a été pensée pour que le joueur ressente une véritable montée en puissance au fur et à mesure de sa progression. Pour réaliser les maps, j'ai utilisé le logiciel Tiled, qui permet de structurer les niveaux en différentes couches séparant les éléments visuels, les collisions et les zones interactives. Cette organisation était essentielle car le projet est devenu progressivement beaucoup plus complexe avec l'ajout des ennemis, des boss, des téléporteurs, des checkpoints, des interactions et du multijoueur. Chaque zone devait rester claire techniquement tout en étant suffisamment détaillée visuellement pour renforcer l'immersion du joueur.

L'un des aspects les plus importants du travail sur les maps concernait la cohérence entre le gameplay et l'environnement. Chaque région du jeu possède une ambiance propre mais aussi une fonction précise dans la progression du joueur. Putrid Fen, par exemple, représente une zone marécageuse sombre et oppressante qui pousse le joueur à observer davantage son environnement et à mieux maîtriser les déplacements de base. Cette région marque également une étape importante du jeu puisque c'est ici que le joueur débloque le double saut. Cette capacité change immédiatement la manière d'aborder l'exploration :

certaines plateformes auparavant inaccessibles deviennent atteignables et le joueur commence à ressentir sa progression.

Fungilia Maze possède une approche totalement différente. Cette zone mise davantage sur la verticalité, les déplacements rapides et les plateformes plus techniques. L'ambiance repose sur un univers organique rempli de champignons géants et de couleurs beaucoup plus vives que dans les autres régions. Le joueur y débloque le dash, une compétence qui accélère fortement le rythme du gameplay et transforme les combats en les rendant beaucoup plus dynamiques. Toute la structure de cette map a été pensée autour de cette capacité afin d'encourager les déplacements rapides et les enchaînements plus fluides.

Par la suite, Amethyst Citadel adopte une approche plus orientée exploration. Cette région demande une meilleure maîtrise des mécaniques du jeu. L'environnement utilise principalement des couleurs sombres afin de créer un contraste fort avec les autres zones du monde. L'objectif était que chaque nouvelle région donne au joueur l'impression de découvrir un nouvel univers plutôt qu'une simple variation esthétique des niveaux précédents.

Mount Emberfall occupe quant à lui une place centrale dans l'aventure. Le volcan sert d'abord de zone tutorielle avant de devenir progressivement l'une des régions les plus dangereuses du jeu. Cette évolution était volontaire : le joueur revient plus tard dans une zone qu'il pensait maîtriser mais découvre alors un environnement beaucoup plus hostile exploitant toutes les capacités débloquées précédemment.

J'ai effectué de nombreux tests afin de garantir des déplacements fluides et cohérents dans toutes les zones du jeu. Chaque plateforme, chaque passage étroit et chaque interaction devait fonctionner correctement aussi bien en solo qu'en multijoueur. Les maps devaient également intégrer l'ensemble des autres systèmes du projet : ennemis, boss, drops, sauvegardes, shop, mini-jeux et événements liés au gameplay. Une attention particulière a donc été portée à l'intégration technique de chaque zone afin d'éviter les conflits entre les différents systèmes.

Enfin, les environnements ont énormément évolué au cours du développement. Les premières versions servaient principalement à tester les déplacements et les collisions. Progressivement, les maps sont devenues beaucoup plus vivantes grâce à l'ajout de décors détaillés, d'arrière-plans, de zones cachées, de systèmes interactifs et d'éléments narratifs.

Les conflits rencontrés

Les systèmes du projet étant fortement liés entre eux, chaque nouvelle mécanique pouvait potentiellement casser une autre partie du jeu. Une grande partie de mon rôle consistait donc à analyser les problèmes, comprendre leur origine puis proposer des solutions permettant de conserver un projet stable et cohérent malgré l'augmentation constante de sa complexité.

L'un des principaux défis rencontrés concernait les merges Git et les conflits entre branches. Le projet étant développé en parallèle par plusieurs membres du groupe, il arrivait régulièrement que différentes modifications touchent les mêmes fichiers au même moment. Ces conflits étaient particulièrement fréquents dans les fichiers centraux du jeu comme `main_game.py`, qui regroupait une grande partie des systèmes importants. Après certains merges, il pouvait arriver que des fonctionnalités cessent complètement de fonctionner : collisions cassées, systèmes d'interaction supprimés, animations désynchronisées ou encore mécaniques réseau devenues incompatibles.

Une grande partie du travail consistait alors à analyser précisément les différences entre les versions afin d'identifier quelles parties du code avaient été modifiées ou écrasées. Dans

plusieurs situations, il a fallu reconstruire manuellement certaines portions de code afin de fusionner correctement deux systèmes développés séparément. Cela demandait non seulement une bonne compréhension de l'architecture globale du projet mais aussi une vision claire des dépendances entre les différents systèmes du jeu.

Le multijoueur représentait également une source importante de bugs complexes. Beaucoup de problèmes apparaissaient uniquement lorsque deux joueurs étaient connectés simultanément. Certaines actions fonctionnaient parfaitement en solo mais provoquaient des désynchronisations en réseau. Par exemple, il arrivait que des objets apparaissent sur l'écran d'un joueur mais pas sur celui de l'autre, que certains ennemis soient morts pour un joueur mais encore vivants pour l'autre ou encore que des récompenses soient dupliquées lors de leur récupération.

Afin de résoudre ces problèmes, j'ai dû retravailler plusieurs systèmes afin de mieux séparer les responsabilités entre le serveur et les clients. Une grande partie des corrections consistait à faire du serveur l'autorité principale concernant les éléments importants du gameplay. Les drops, les récompenses, les interactions et certaines collisions sont désormais validés côté serveur avant d'être synchronisés aux clients. Cette approche a permis de réduire fortement les incohérences entre les deux instances du jeu.

Les systèmes d'inventaire et d'équipement ont eux aussi demandé de nombreuses corrections. Avec l'ajout des armures, des fragments d'épée et des objets consommables, plusieurs problèmes sont apparus progressivement. Certains objets pouvaient être équipés plusieurs fois, certaines images ne s'affichaient pas correctement ou provoquaient des crashs, et certaines récompenses obtenues via le shop ou les boss ne se synchronisaient pas correctement avec l'inventaire du joueur. J'ai donc dû retravailler la gestion des équipements ainsi que les vérifications empêchant les duplications ou les conflits d'état. Dans un jeu de plateforme, des collisions mal placées peuvent rapidement rendre certaines zones injouables. J'ai donc passé beaucoup de temps à tester les niveaux, corriger les plateformes problématiques et ajuster les zones d'interaction afin de garantir des déplacements fluides et cohérents.

Le système de shop a lui aussi nécessité plusieurs corrections importantes. À certaines étapes du développement, le shop ne fonctionnait correctement qu'au clavier, les interactions avec les PNJ étaient incohérentes ou certains achats pouvaient provoquer des erreurs dans l'inventaire. J'ai donc retravaillé cette fonctionnalité afin de stabiliser la navigation, améliorer la compatibilité manette et corriger les problèmes liés aux équipements achetés dans le magasin. Une difficulté importante du projet venait également du fait que beaucoup de bugs apparaissaient uniquement après plusieurs heures de développement ou après l'ajout de nouvelles fonctionnalités. Il ne suffisait donc pas simplement de corriger les erreurs visibles immédiatement. Une grande partie du travail consistait à anticiper les futurs conflits possibles entre les différents systèmes afin de rendre l'architecture du projet plus robuste et plus flexible.

3.4.3 Thibault Ansart : Mini-jeux et Site Web

Mini-jeux

Afin d'insuffler une véritable profondeur mécanique au gameplay et d'enrichir l'expérience globale de l'utilisateur, l'équipe de développement de Reach of Vespers a conçu les puzzles et les mini-jeux non pas comme de simples divertissements secondaires ou des artifices de remplissage, mais comme une composante essentielle, organique et gratifiante de l'aventure. Dans le cadre d'un Metroidvania, genre souvent caractérisé par son inten-

sité constante, ses combats exigeants et son exploration labyrinthique parfois source de tension, ces épreuves ludiques ont pour objectif premier d'offrir une respiration bienvenue aux joueurs.

En rompant temporairement avec le rythme effréné des phases de plateforme traditionnelles, ces intermèdes permettent de modifier la dynamique psychologique de la session de jeu, en déplaçant le curseur de l'habileté motrice vers une stimulation plus intellectuelle, mettant ainsi l'accent sur la réflexion posée et la déduction logique plutôt que sur la pure dextérité technique ou les réflexes instantanés.

Ces ajouts constituent ainsi une part conséquente de l'expérience de jeu globale et s'inscrivent pleinement dans la progression du joueur, en offrant des récompenses concrètes et tangibles à chaque complétion. En effet, chaque mini-jeu rapporte 100 améthystes, la monnaie officielle de Reach of Vespers, ouvrant ainsi la voie à des acquisitions stratégiques au sein du shop du jeu. Cette monnaie précieuse, au cœur du système économique du titre, s'avère indispensable pour débloquer des avantages stratégiques majeurs tels que des équipements améliorés, des objets de soin cruciaux, des reliques anciennes ou encore des éléments cosmétiques exclusifs qui facilitent et enrichissent la progression ultérieure du joueur.

Les mini-jeux ont pour objectif de ne pas s'arrêter à la simple réalisation, mais d'aller plus loin dans l'utilité du gameplay, en s'intégrant de manière fluide et directe dans l'économie globale du titre et en récompensant toujours concrètement l'effort intellectuel fourni. Néanmoins, ils gardent également leur distance avec le cœur du jeu, demeurant entièrement optionnels afin de ne pas contrarier les joueurs de Reach of Vespers qui préfèrent se concentrer exclusivement sur l'exploration de la carte et le dynamisme des combats.

De surcroît, chaque défi est méticuleusement contextualisé pour préserver l'originalité de l'expérience et bannir définitivement toute sensation de répétitivité mécanique. Les énigmes suivent scrupuleusement le fil rouge de la narration et s'adaptent de manière organique au biome spécifique dans lequel se trouve le protagoniste. Par exemple, lorsque l'exploration mène le joueur dans un environnement souterrain ou sylvestre saturé de champignons, le puzzle associé exigera d'interagir intelligemment avec des champignons colorés selon une séquence rythmique précise, ou encore de manipuler des ingrédients mystiques et des herbes médicinales sur une planche de cuisine de fortune. Cette approche garantit que le gameplay reste intimement lié au lore de l'univers, renforçant à chaque instant le sentiment d'immersion et de cohérence narrative.

Sur le plan artistique et visuel, la conception de ces mini-jeux a également évolué au fil du développement. Si la première intention de l'équipe était d'opter pour une esthétique pixel art, il est rapidement apparu que ce choix aurait introduit une rupture de cohérence avec l'univers sombre, dense et soigneusement construit de Reach of Vespers. L'équipe J-1 a donc pris la décision de retravailler entièrement la direction artistique de ces séquences, en les orientant vers un style cartoon plus réaliste, toujours ancré dans la dark fantasy typique de l'univers du jeu, garantissant ainsi une immersion constante et une unité visuelle préservée de bout en bout.

Ce parti pris artistique crée par ailleurs un effet de mise en abyme particulièrement captivant, instaurant un véritable sentiment de "jeu dans le jeu" : dès qu'un puzzle est activé, l'action principale du monde extérieur se fige instantanément, plongeant l'environnement dans une pause salvatrice permettant une résolution sereine et sans stress temporel. À cet instant, l'affichage adopte systématiquement un format carré très caractéristique, dont le changement soudain de cadrage et de géométrie visuelle fait office de signalétique intuitive, indiquant immédiatement au joueur qu'il quitte le plan de l'action

physique pour pénétrer dans un espace confiné dédié à l'esprit et à la logique. Durant ces phases, l'utilisateur conserve par ailleurs un contrôle absolu et ergonomique sur l'interface, bénéficiant de la liberté totale de réinitialiser l'épreuve à son état initial en cas d'erreur, de la quitter instantanément pour y revenir ultérieurement, ou de valider ses choix lorsqu'il estime avoir trouvé la solution adéquate.

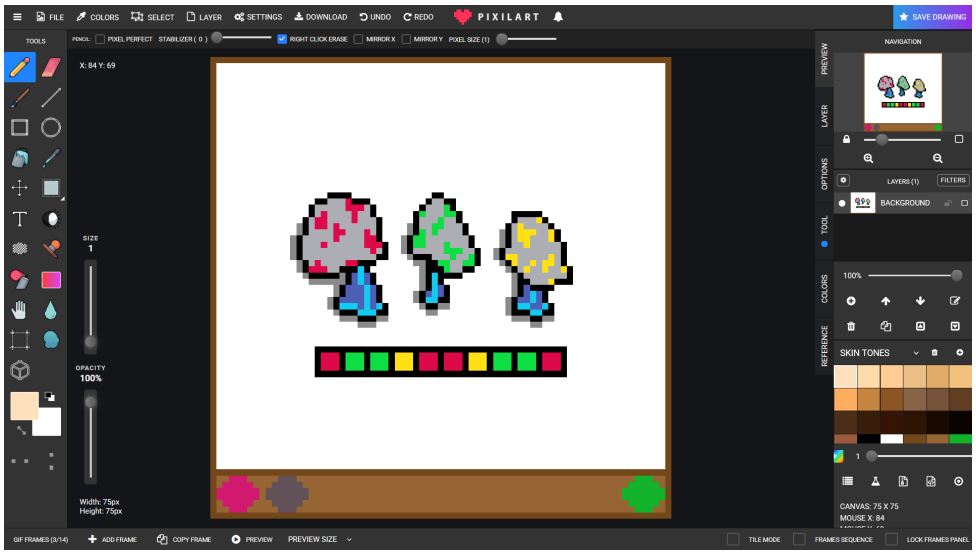
En coulisses, la chaîne de production et d'intégration de ces modules témoigne d'une méthodologie technique à la fois rigoureuse et artisanale. Les assets visuels et les illustrations qui composent ces puzzles sont entièrement façonnés à la main dans un premier temps en pixel art, en conservant scrupuleusement une ambiance dark fantasy à chaque élément graphique, avant d'être réinterprétés afin de produire les modèles cartonnés définitifs. Sur le plan de la programmation, le processus respecte des étapes strictes et ordonnées : avant toute intégration visuelle, les puzzles font l'objet d'un prototypage algorithmique approfondi à l'aide de scripts codés en Python, ce qui permet à l'équipe de tester la difficulté stratégique de l'énigme, sa fluidité d'exécution et son intérêt ludique intrinsèque. Une fois la mécanique de jeu validée sur le plan logique, les visuels finaux sont exportés sous la forme de fichiers PNG et intégrés au moteur du jeu.

Enfin, soucieuse de prolonger l'immersion narrative et d'engager sa communauté de joueurs bien au-delà des frontières de l'application principale, l'équipe a également développé un mini-jeu totalement inédit et exclusif, accessible directement depuis le site web officiel du projet. Ce module externe sert à la fois de passerelle interactive pour les nouveaux venus et de divertissement complémentaire pour les fans de la première heure, ancrant durablement l'univers de Reach of Vespers dans un écosystème global et transmédia, et témoignant de la volonté de l'équipe J-1 de construire une expérience qui dépasse les seules frontières du jeu lui-même.

Sur le plan de l'architecture technique, les mini-jeux s'intègrent au sein du projet selon une organisation rigoureuse et structurée. L'ensemble des classes associées aux mini-jeux est regroupé dans un fichier dédié, `MiniGame.py`, développé selon les principes de la programmation orientée objet. Ce fichier centralise toute la logique propre à chaque séquence de jeu, et les instances sont appelées depuis le fichier principal `main_game.py` sous la forme de références numérotées, allant de `mg-1` à `mg-5`, permettant ainsi une gestion claire et modulaire de chaque module. `MiniGame.py` interagit également directement avec les fichiers liés au joueur, afin de lui attribuer les 100 améthystes dès lors qu'un mini-jeu est réussi, assurant ainsi une connexion fluide entre la logique des énigmes et le système de progression global du personnage.

Sur le plan du chargement et de l'intégration au sein du moteur de jeu, une réflexion approfondie a guidé les choix d'implémentation retenus. Initialement, les mini-jeux étaient conçus pour se charger en dehors du jeu principal, sur une fenêtre séparée. Cette approche soulevait cependant un problème de performance significatif : le lancement d'un nouveau processus parallèlement au processus principal du jeu engendrait une charge système inutile et peu optimisée. La décision a donc été prise de charger les mini-jeux directement au sein de l'environnement du jeu. Dans un second temps, lorsqu'un mini-jeu est appelé, le jeu en arrière-plan bascule dans un état spécifique similaire à celui de la pause, afin de figer l'action principale le temps de la résolution de l'énigme. Pour des raisons d'optimisation supplémentaires, l'état de pause existant a finalement été retenu comme état définitif pour gérer ce basculement, évitant ainsi la création d'un état redondant et allégeant d'autant la charge de traitement.

Concernant l'interface de contrôle au sein des mini-jeux, la lisibilité et l'ergonomie ont été placées au cœur des décisions de conception. Afin de permettre au joueur de naviguer

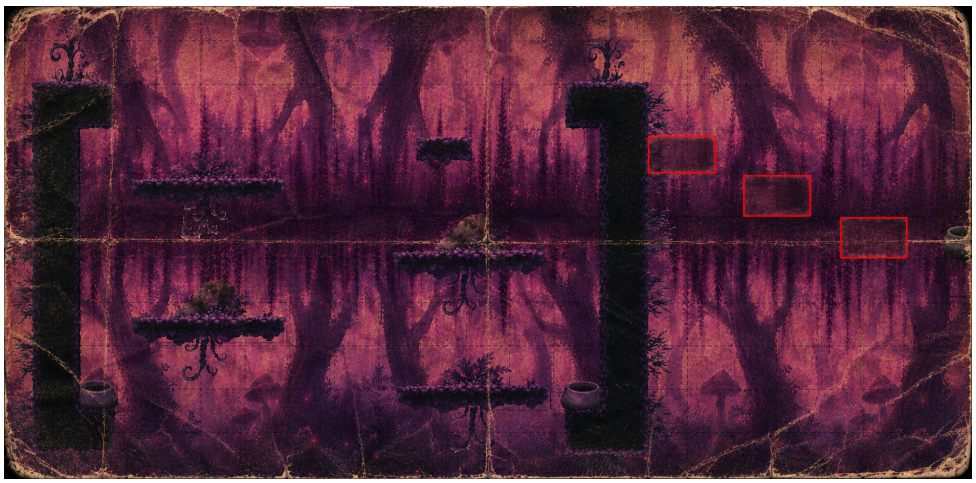


intuitivement entre les différentes actions disponibles (quitter, recommencer ou valider), le choix s'est porté sur un système de touches de couleur explicites : rouge pour quitter, brun pour recommencer, et vert pour valider. Ce système de codage chromatique avait pour but d'être immédiatement compréhensible, mais a néanmoins posé des difficultés dans sa première incarnation en pixel art, où la lisibilité de ces indications restait insuffisante pour garantir une compréhension immédiate de la part du joueur. Ce constat a d'ailleurs constitué l'un des facteurs déterminants dans la décision de basculer vers le style cartoon, bien plus adapté à la communication visuelle de ces interactions. Des zones de détection ont par ailleurs été intégrées sur les éléments interactifs tels que les champignons et les boutons, assurant une précision satisfaisante dans la prise en compte des actions du joueur.

Sur le plan du processus de création des assets visuels, une méthodologie en deux temps a été adoptée. Chaque illustration constitutive des mini-jeux a d'abord été dessinée à la main en pixel art sur la plateforme Pixilart.com, en conservant à chaque étape une atmosphère cohérente avec l'univers dark fantasy de Reach of Vespers. Ces créations ont ensuite été réinterprétées et retravaillées afin de produire les modèles dans le style cartoon sombre constituant le rendu visuel final. Avant toute intégration visuelle, les mécaniques de

jeu ont également fait l'objet d'un prototypage algorithmique en Python sur des branches à part, permettant de valider la difficulté, la fluidité et l'intérêt ludique de chaque énigme avant d'y associer ses visuels définitifs, exportés sous forme de fichiers PNG.

Enfin, sur le plan de la cohérence narrative, les énigmes textuelles ont été pensées pour s'inscrire, de près ou de loin, dans l'univers de Reach of Vespers. Ainsi, les solutions retenues *"silence, ombre, et d (le dé)"* participent toutes trois à l'atmosphère du jeu : le silence et l'ombre s'intègrent naturellement à l'ambiance sombre et crépusculaire du titre, tandis que le dé, bien que légèrement plus éloigné de l'univers narratif, évoque le hasard, l'injustice et le jeu, des thématiques qui s'inscrivent pleinement dans le cadre des mini-jeux. Le mini-jeu `mg-2`, quant à lui, va plus loin que la simple récompense en améthystes : en le réussissant, le joueur obtient l'affichage d'une carte vieillie, générée à partir du TMX de la zone cachée correspondante, révélant l'emplacement d'un coffre supplémentaire dans la carte Mushroom. Durant cet affichage, le jeu demeure en état de pause afin d'éviter tout conflit de traitement, et le joueur peut passer la carte en cliquant sur n'importe quel bouton.



Le développement des mini-jeux a été parsemé de plusieurs obstacles techniques et organisationnels, dont la résolution a nécessité rigueur, méthode et adaptabilité. Le premier défi majeur a été celui de gérer les mini-jeux au sein du jeu en lui-même. Comme évoqué précédemment, la première approche consistant à charger les mini-jeux dans une fenêtre externe s'est rapidement révélée inadaptée, générant un second processus concurrent au processus principal et dégradant les performances globales. La solution a résidé dans le rapatriement complet des mini-jeux à l'intérieur du jeu, en tirant parti de l'état de pause existant comme socle d'intégration, évitant ainsi toute duplication de logique et tout surcoût système inutile.

Le second obstacle, et sans doute le plus long en temps, a été celui du merge final. Du fait d'une avancée simultanée et intensive de l'ensemble des membres de l'équipe, la phase d'intégration du code a généré un volume important de merge conflicts. Un des conflits concernait la fonction `open_chest` dans le fichier `main_game.py`, qui avait été modifiée de manière indépendante dans d'autres branches merges. Un second problème, particulièrement difficile à identifier, résidait dans une incohérence de nommage entre les identifiants des mini-jeux définis dans le fichier TMX (référencés sous la forme `mg-1`, `mg-2`, jusqu'à `mg-5`) et les noms de variables utilisés dans `main_game.py` pour récupérer les classes correspondantes, qui suivaient quant à eux la convention `mj1`, `mj2`, jusqu'à `mj5`. Cette divergence mineure a nécessité un temps de débogage considérable avant d'être identifiée, puis corrigée.

Sur le plan des tests, une méthode de travail spécifique a été mise en place afin de faciliter le développement itératif des mini-jeux. Une variable `MINI_JEU_COURANT` permettait de désigner le mini-jeu à tester, et la touche *"n"* du clavier avait été mappée pour le lancer directement, à l'image de la touche *"p"* déjà existante pour la téléportation entre niveaux. Ce dispositif a permis de tester chaque séquence de manière rapide et isolée, sans avoir à parcourir l'intégralité du jeu pour y accéder.

Enfin, certains mini-jeux initialement envisagés ont dû être abandonnés en cours de développement. Si des idées visuelles avaient émergé, elles ne s'accompagnaient pas d'un gameplay suffisamment concret et stratégique pour justifier leur intégration dans un état satisfaisant. La décision a donc été prise de ne pas les inclure dans la version finale, afin de préserver le niveau de qualité et de cohérence attendu pour l'ensemble des séquences proposées au joueur.

Site web

Le site web de Reach of Vespers a été conçu pour être le prolongement direct et naturel de l'univers du jeu, bien au-delà d'une simple vitrine de présentation. Consultable à l'adresse suivante : `https://ansartthibault.github.io/ReachOfVespers/`, il constitue une extension transmédia à part entière, pensée pour ancrer durablement l'univers du projet dans un écosystème global et accessible à tous, joueurs comme néophytes.

L'identité visuelle du site repose sur un contraste fort et assumé entre le noir et le violet. Le noir sert d'émulation directe à l'ambiance sombre et pesante du jeu, tandis que le violet évoque une puissance lumineuse latente, capable de virer au rose éclatant selon les contextes. Cette lumière sacrée, que l'on devine en arrière-plan à travers les différentes pages, renforce l'aspect mystique et crépusculaire du projet, en cohérence totale avec la direction artistique de Reach of Vespers. L'UI et l'UX ont par ailleurs été pensées pour s'adapter aussi bien au format PC qu'au format mobile, bien que la consultation sur

ordinateur soit recommandée afin de profiter pleinement de la mise en page et des effets visuels dans leur intégralité.

La direction artistique s'exprime également par le son. Lors du clic sur le logo *"ROV"* de la page d'accueil, une balise audio déclenche un effet sonore puissant, dont l'objectif est de maintenir le visiteur en immersion totale, transformant une simple interaction de navigation en une véritable transition, semblable à l'ouverture d'un portail vers l'univers du jeu.

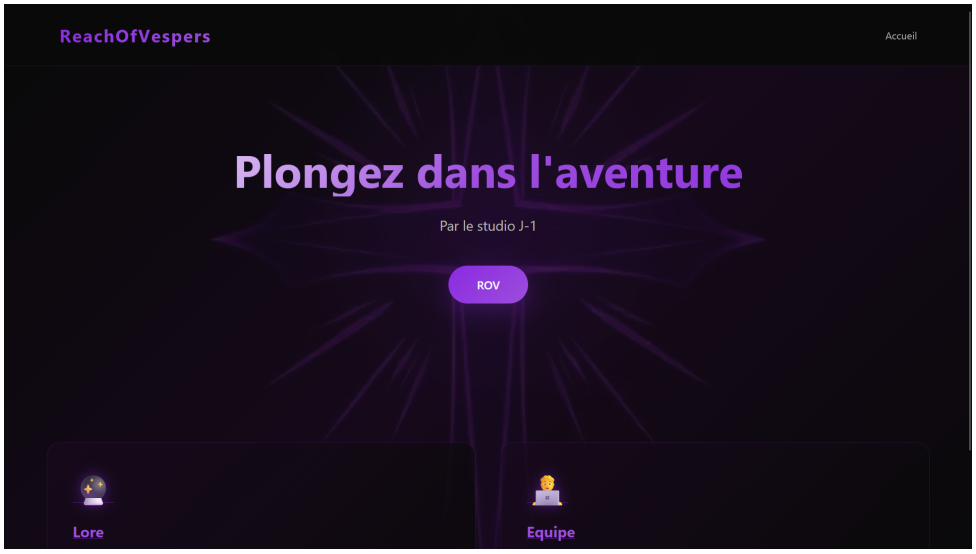
En haut de chaque page, une barre de navigation persistante permet de revenir à l'accueil à tout moment, quel que soit le niveau de défilement de l'utilisateur. Le titre du jeu y est systématiquement affiché à gauche, garantissant un rappel contextuel constant et ancrant visuellement chaque page dans l'identité du projet. Pour accentuer l'interactivité et enrichir l'expérience de navigation, le passage de la souris sur cette barre de navigation déclenche plusieurs effets simultanés : un agrandissement visuel, l'apparition d'un filet lumineux violet animé, ainsi qu'une mise en mouvement du titre. Des effets similaires ont été appliqués au bouton *"Accueil"* afin de confirmer visuellement l'interaction à l'utilisateur et d'assurer une cohérence comportementale sur l'ensemble des éléments interactifs du site.

Le site se compose de plusieurs pages aux rôles bien distincts :

— **La page Lore** a pour vocation de résumer l'intrigue du jeu et de stimuler la curiosité des visiteurs, en y développant l'univers, les biomes et les entités qui peuplent Reach of Vespers. Le contenu y est volontairement dense sur le plan narratif, tout en préservant les surprises liées à l'expérience de jeu elle-même.

— **La page Équipe** présente quant à elle les cinq membres de la Team J-1 (Clément, Jules, Thibault, Thomas et Tristan) en détaillant les rôles respectifs de chacun au sein du projet.

— **La page ROV** permet à l'utilisateur de télécharger les fichiers nécessaires, à savoir le jeu lui-même ainsi que le rapport de soutenance, et propose également un mode d'emploi de téléchargement pour accompagner les nouveaux venus dans leur prise en main.

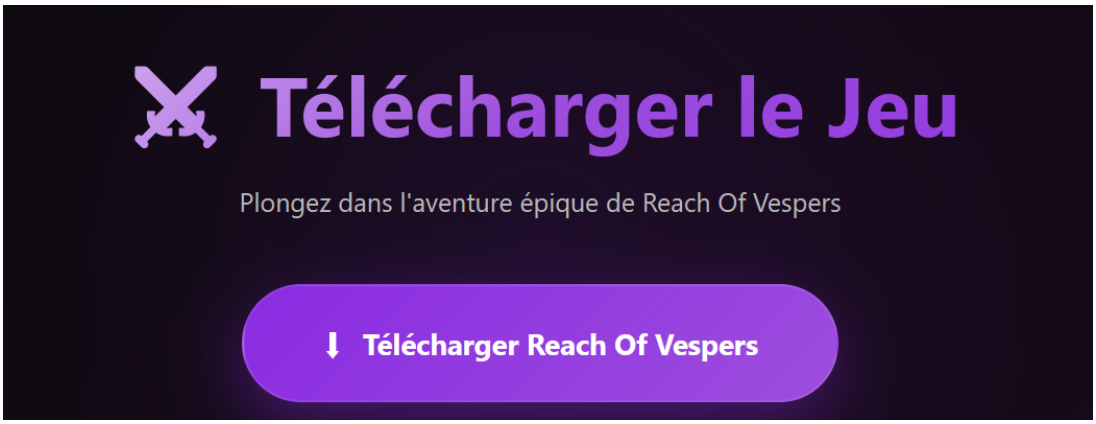


Le site est actuellement hébergé sur GitHub Pages et repose sur les technologies standards du développement web, mobilisées chacune selon leur domaine de compétence propre :

— **Le HTML** constitue le socle fondamental du site, utilisé comme langage de balisage pour poser les fondations structurelles et définir les conteneurs de chaque page.
— **Le CSS** prend en charge l'intégralité de la mise en forme et de l'esthétique, organisé via l'utilisation d'identifiants et de classes. La grande majorité des animations fluides présentes sur le site sont gérées par des règles `@keyframes`, offrant une maîtrise précise du comportement visuel de chaque élément animé dans le temps.

— **Le JavaScript** apporte la dimension dynamique nécessaire aux interactions les plus complexes, notamment la gestion du portail sonore sur la page d'accueil et les transitions entre les différentes pages du site.

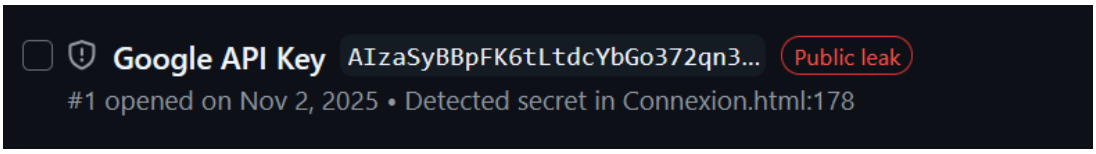
Sur le plan des animations de la barre de navigation, chaque lettre composant le titre du jeu a été isolée individuellement dans une balise `span`, à laquelle des animations CSS spécifiques ont été associées. Cette approche granulaire permet de piloter indépendamment le comportement de chaque caractère, offrant ainsi un effet de mise en mouvement du titre fluide et maîtrisé au survol de la barre de navigation, en cohérence avec l'identité visuelle dynamique et vivante que le site cherche à incarner.



Problèmes rencontrés et solutions apportées : Le développement du site web a été marqué par plusieurs problèmes d'orientations et obstacles techniques, dont certains ont nécessité une réflexion importante. Le premier problème a été d'ordre conceptuel. L'idée d'intégrer un shop directement sur le site web avait initialement été envisagée, mais s'est rapidement révélée inadaptée au contexte du projet. Reach of Vespers étant un projet étudiant annuel et non un jeu destiné à être commercialisé sous cette forme, l'ajout d'un shop n'avait pas de justification fonctionnelle réelle et aurait introduit une complexité inutile sans apporter de valeur concrète à l'expérience proposée. Cette fonctionnalité a donc été abandonnée.

Dans la continuité de cette réflexion, l'intégration d'une connexion avec Firebase avait également été envisagée, notamment pour permettre une forme de gestion de comptes utilisateurs. Si cette implémentation était techniquement envisageable, elle a soulevé deux problèmes majeurs. D'une part, les identifiants de connexion se sont retrouvés exposés publiquement sur le dépôt GitHub, constituant une fuite de données potentiellement problématique. D'autre part, la connexion à un compte Gmail depuis le jeu ne présentait

aucune utilité concrète dans le cadre du projet tel qu'il était défini, cette fonctionnalité n'ayant de sens que dans le contexte d'un shop qui avait lui-même été abandonné. L'intégration Firebase a donc été écartée à son tour.



Le second obstacle technique a concerné la gestion du son. L'effet sonore déclenché au clic sur le logo de la page d'accueil, bien que fonctionnel dans la majorité des cas, s'est révélé soumis à des comportements variables selon les navigateurs utilisés. Certains navigateurs appliquent en effet des politiques restrictives concernant la lecture automatique ou déclenchée de sons, pouvant empêcher la balise audio de fonctionner correctement sans intervention préalable de l'utilisateur. Après analyse, il est apparu que le problème se résout dans la grande majorité des situations dès lors que l'utilisateur autorise les sorties son du site web dans les paramètres de son navigateur. La balise audio fonctionne donc correctement dans ce cadre, et ce comportement variable a été accepté comme une contrainte inhérente aux environnements de navigation, sans nécessiter de solution de contournement supplémentaire.

3.4.4 Clément Genty : Monstres/Boss, Classe Joueur, Tutoriel, Système de sauvegarde et Marketing

Monstres/Boss

L'objectif central de mon travail sur Reach of Vespers était de concevoir et d'animer l'écosystème hostile du jeu, un élément fondamental pour tout jeu d'action-plateforme. La viabilité de notre boucle de gameplay reposait sur la capacité du moteur à gérer de multiples entités simultanément, qu'il s'agisse de monstres standards (comme les Golems de pierre, les Nécromanciens ou les Oracles sombres) ou d'affrontements majeurs contre des Boss de fin de niveau (Magmaroth, Wraith, Malakor). Il ne s'agissait pas simplement d'afficher des images à l'écran, mais de créer une véritable architecture logique capable de soutenir des comportements variés et autonomes, tout en garantissant des performances optimales. La difficulté supplémentaire résidait dans la conception des Boss. Ces entités devaient non seulement posséder des statistiques hors normes et une interface dédiée, comme une jauge de vie massive ancrée en haut de l'écran, mais aussi exécuter des schémas d'attaque complexes incluant des lancers de projectiles, des altérations de zone et des invocations. Enfin, l'intégralité de ces comportements devait être parfaitement répliquée et synchronisée pour les deux joueurs lors des sessions en mode multijoueur local ou en ligne, imposant une rigueur absolue dans la gestion de l'état des entités.

Pour garantir la flexibilité et la maintenabilité du code, j'ai structuré mon travail autour d'une architecture orientée objet en m'appuyant sur les modules de sprites natifs de notre moteur. J'ai d'abord développé une classe mère globale chargée d'encapsuler la physique, la gestion des collisions environnementales, la détection des joueurs et l'actualisation des frames d'animation.

Au-delà de la pure logique de programmation, mon rôle s'est étendu à la direction technique et artistique. Bien que notre base de ressources graphiques ait été sourcée depuis la

bibliothèque publique CraftPix.net, ces paquets génériques ne couvraient pas l'exhaustivité de nos besoins mécaniques. J'ai par exemple été contraint de recréer manuellement des séquences d'animation intermédiaires totalement absentes des fichiers d'origine, notamment pour permettre l'exécution fluide d'attaques de mêlée ou de lancers de projectiles en pleine course, sans briser l'inertie du mouvement. Dans cette même démarche d'intégration visuelle, j'ai opéré une recoloration complète des feuilles de sprites de l'Oracle Sombre, dont les teintes par défaut entraient en dissonance avec l'atmosphère du jeu, afin de l'intégrer harmonieusement dans la palette chromatique très spécifique de notre biome fongique.

L'intelligence artificielle de ces adversaires est régie par le principe mathématique de la machine à états finis. Chaque monstre possède une variable d'état stricte qui dicte son comportement à l'instant T : patrouille, poursuite, attaque, encaissement de dégâts, ou mort. À chaque itération de la boucle de rafraîchissement du jeu, l'algorithme évalue l'environnement pour déterminer si une transition d'état est nécessaire. Par exemple, lors de la phase de patrouille, un algorithme de balayage liste les joueurs présents sur la carte, vérifie s'ils sont ciblables, calcule la distance absolue et verrouille le plus proche.

Un autre défi technique concernait la précision des zones de collision. Dans un jeu aux animations fluides, les dimensions visuelles d'un personnage changent drastiquement selon qu'il soit au repos ou en pleine attaque. L'utilisation de rectangles de collision classiques provoquait des comportements erratiques. Pour pallier cela, j'ai implémenté un algorithme de masquage dynamique générant une hitbox basée exclusivement sur les pixels non transparents de la frame en cours. J'ai ensuite mathématiquement ancré toutes les entités par le centre de leur base inférieure, garantissant que leurs pieds restent perpétuellement alignés avec le sol.

Le défi technique le plus critique de mon module est apparu lors des phases de test du mode multijoueur, révélant une faille de conception dans notre système de synchronisation. Visuellement, le bug était catastrophique sur la machine du client : lors du combat final, le Boss adoptait soudainement le comportement d'un monstre de base avec une minuscule barre de vie, tandis qu'un Golem aléatoire situé à l'autre bout du niveau devenait

gigantesque et héritait de la vie du Boss. Sur la machine de l'hôte, en revanche, tout fonctionnait parfaitement.

En analysant le flux des paquets de données échangés sur le réseau, j'ai identifié que le problème provenait de la méthode d'indexation de notre liste d'ennemis. Le serveur transmettait l'état des monstres sous la forme d'un tableau de données, que le client appliquait de manière séquentielle, index par index. Le Boss étant toujours généré en dernier, il occupait la dernière position de cette liste. Cependant, si le joueur hôte tuait un monstre standard en cours de route, ce dernier était supprimé de la mémoire du serveur, décalant logiquement le Boss d'un cran vers le début de la liste. Le client, recevant cette nouvelle structure de données, appliquait les statistiques du Boss à l'entité locale correspondant à ce nouvel index, qui s'avérait être un Golem encore en vie sur sa machine.

Pour résoudre cette désynchronisation critique, j'ai dû entièrement repenser l'algorithme d'identification réseau en abandonnant la logique d'indexation séquentielle. J'ai introduit un système d'Identifiant Unique (UID) pour chaque entité. Puisqu'il est mathématiquement impossible dans notre génération de niveau que deux monstres apparaissent exactement au même pixel, j'ai utilisé les coordonnées d'apparition originelles de chaque ennemi pour forger une clé alphanumérique inaltérable dès sa création. Du côté du client, j'ai réécrit l'algorithme de réception : le jeu extrait désormais l'UID du paquet de données, parcourt ses propres entités locales pour trouver la correspondance exacte, puis applique les mises à jour. Ce correctif a rendu la synchronisation multijoueur parfaitement résiliente et a stabilisé les combats de Boss.

La nature même de mon pôle m'a imposé de collaborer étroitement avec l'ensemble des autres développeurs pour interconnecter nos systèmes. Du côté du Level Design, mes algorithmes de physique ont dû être interfacés avec le moteur de rendu de carte en tuiles. Les entités que je programmais n'avaient pas de coordonnées codées en dur, mais étaient instanciées dynamiquement en lisant les calques d'apparition placés par mes collègues dans le logiciel de création de cartes. Ma fonction de gestion de la gravité extrayait directement la liste des plateformes générée par l'environnement pour calculer les collisions en temps réel.

En matière de réseau, j'ai travaillé de concert avec l'architecture multijoueur. Mes classes d'ennemis ont été équipées de méthodes de sérialisation complexes, capables de compiler l'intégralité de leur état cognitif dans des dictionnaires de données compressés. Ces paquets étaient ensuite confiés au gestionnaire réseau pour être envoyés et interprétés par le second joueur, exigeant que mes monstres soient capables d'alterner dynamiquement entre une intelligence artificielle autonome et un mode passif piloté par le serveur.

Le raccordement avec le système d'économie a également été une étape majeure. J'ai conçu la séquence de mort des adversaires de manière à ce qu'elle déclenche un algorithme probabiliste communiquant avec le gestionnaire de butin. À l'instant exact où les points de vie atteignent zéro, mon code interroge une table de probabilités pour générer d'éventuelles récompenses (monnaie, bonus de statistiques). Cette donnée est ensuite récupérée par le système d'inventaire du joueur pour mettre à jour son portefeuille et lui permettre d'interagir avec la boutique du jeu.

Enfin, l'immersion sonore a requis une synchronisation minutieuse avec le moteur audio. Pour donner du poids aux affrontements, j'ai inséré des déclencheurs de bruitages directement au cœur de ma machine à états. Plutôt que de jouer des sons de manière arbitraire, les effets sonores de pas, de coups d'épée ou de souffrance sont appelés mathématiquement à la milliseconde près, lors de la transition d'un état à un autre ou lors de l'activation d'une frame spécifique de l'animation, garantissant une cohérence parfaite

entre le travail sonore de mon équipe et le retour visuel à l'écran.

Classe Joueur

La classe Joueur représente le cœur névralgique de l'expérience utilisateur dans Reach of Vespers. Le développement d'un jeu de plateforme et d'action exige une maniabilité irréprochable, souvent qualifiée de "pixel-perfect". Mon objectif principal était de concevoir un contrôleur de personnage fluide, réactif et évolutif, capable d'exécuter un large panel de mouvements complexes tels que le double saut, l'esquive rapide (dash) et la glissade sur les murs (wall-slide). Au-delà de la simple motricité, cette classe devait également centraliser les mécaniques de combat de mêlée, la gestion de l'équipement, ainsi que l'évolution des statistiques du personnage au fil de l'aventure via un système de fragments d'épée et d'améliorations temporaires. Le défi consistait à intégrer toutes ces composantes dans une seule et même entité logique sans créer de conflits d'états, tout en garantissant une compatibilité totale avec les systèmes de sauvegarde et de multijoueur.

Pour architecturer les mouvements du personnage, j'ai développé un moteur physique personnalisé (custom physics) basé sur la résolution des collisions par séparation d'axes. Plutôt que de déplacer le joueur en diagonale et de risquer des chevauchements imprévisibles avec les angles des plateformes, l'algorithme applique d'abord le vecteur de vélocité horizontale, teste les collisions avec l'environnement et corrige la position X si nécessaire. Seulement après cette étape, le vecteur de gravité (vélocité verticale) est appliqué et vérifié de la même manière. Cette méthode mathématique garantit que le joueur ne reste jamais accroché dans les arêtes géométriques de la carte. De plus, j'ai implémenté un système de friction dynamique : lorsque le joueur relâche les touches de direction, la vélocité n'est pas mise à zéro instantanément, mais décroît par soustraction d'un coefficient de friction, offrant une sensation d'inertie naturelle.

La gestion des états d'animation et de physique repose sur un arbre de priorités strict. Par exemple, l'état d'esquive (dash) est gouverné par un chronomètre interne indépendant. Lorsqu'il est déclenché, il écrase temporairement la gravité et force une vélocité horizontale maximale pendant un nombre précis de frames.

Afin de gérer les objets consommables (potions, gemmes) sans corrompre les statistiques de base du personnage, j'ai conçu un algorithme d'altération d'état (buffs) dynamique. Lorsqu'un consommable est utilisé, un dictionnaire contenant la statistique visée, la valeur de l'augmentation et la durée de l'effet est ajouté à un tableau d'altérations actives. À chaque itération de la boucle principale, la fonction de mise à jour décrémente les chronomètres de chaque élément de ce tableau. Dès qu'un chronomètre atteint zéro, l'algorithme soustrait mathématiquement la valeur du bonus à la statistique du joueur et supprime l'entrée du tableau. Ce système permet d'empiler plusieurs effets simultanément en toute sécurité. Enfin, les dégâts de l'arme sont calculés dynamiquement par une fonction de mise à jour qui lit le nombre de fragments d'épée ramassés pour appliquer un multiplicateur mathématique sur l'attaque de base.

L'un des défis techniques les plus épineux est survenu lors de l'implémentation de la glissade sur les murs (wall-slide), qui provoquait de graves conflits avec la machine à états de l'animation et le moteur audio. Initialement, lorsque le joueur s'accrochait à un mur, le statut "is_wall_sliding" s'activait, freinant la chute et déclenchant la piste audio de glissade en boucle. Cependant, si le joueur effectuait une esquive (dash) ou se laissait tomber sans sauter, le jeu considérait qu'il n'avait jamais "quitté" le mur d'un point de vue de la collision latérale immédiate. Résultat : le personnage se figeait dans

son animation de glissade tout en tombant dans le vide, et le son de frottement contre la pierre continuait de se jouer indéfiniment, saturant le canal audio.

Pour résoudre cette désynchronisation, j'ai dû concevoir un système de raycasting virtuel (projection de masque). L'algorithme ne se contente plus de vérifier si la hitbox du joueur touche un mur. Au lieu de cela, si le joueur est en l'air et appuie dans une direction, le code génère une copie invisible de sa hitbox de collision, décalée mathématiquement de 2 pixels dans la direction pressée. La glissade n'est validée que si cette projection virtuelle intercepte une plateforme, prouvant l'intentionnalité du mouvement. Pour corriger le problème audio, j'ai découplé la lecture du son de l'état d'animation pur en introduisant un moniteur d'état booléen restrictif. Le code vérifie à chaque frame si l'état exact est "wall_slide". Si oui, il lance la piste audio uniquement si le moniteur indique qu'elle n'est pas déjà en cours. Dès que la machine à états bascule sur n'importe quel autre mouvement (chute libre, saut, esquive, dégâts), un bloc de contrôle dédié intercepte ce changement, force l'arrêt immédiat du flux audio sur ce canal spécifique et réinitialise le moniteur. Cette architecture croisée a permis de rendre la mécanique de glissade et les sauts muraux (wall-jumps) extrêmement fiables.

En tant qu'avatar contrôlé par l'utilisateur, ma classe Joueur agit comme le point de convergence de tous les autres modules du projet. Sur le plan des entrées (inputs), j'ai dû relier mon algorithme de déplacement au dictionnaire de configuration des touches du système global, tout en implémentant une surcouche lisant en temps réel la position des axes analogiques (joysticks) pour rendre le jeu totalement jouable à la manette.

La physique du personnage interagit directement avec le Level Design. Ma fonction de mise à jour appelle en permanence la liste des plateformes générée procéduralement par le gestionnaire de cartes en tuiles (Pyscroll) de mon camarade pour calculer les collisions. Du côté de l'immersion sonore, j'ai synchronisé mon système d'animation avec le module audio : le son des bruits de pas est déclenché par un chronomètre calibré sur la cadence de l'animation de course, et surtout, il récupère dynamiquement les banques de sons du biome actuel (pierre, herbe, boue) via les propriétés du niveau chargé.

L'interconnexion avec les systèmes de combat et d'économie fut également un axe de travail majeur. J'ai exposé des interfaces publiques de prise de dégâts et de recul (bump) que les projectiles, boss et monstres conçus par le reste de l'équipe peuvent appeler lors d'une collision. Ces fonctions intègrent un algorithme d'absorption qui soustrait automatiquement la valeur d'armure du joueur avant d'amputer sa jauge de santé. Enfin, pour le système de sauvegarde, la fonction de sérialisation du Joueur ne se contente pas de sauvegarder ses coordonnées et ses compétences ; elle encapsule et délègue également la sauvegarde de l'objet "Inventaire" codé par un autre membre de l'équipe, assurant ainsi qu'un seul fichier JSON puisse restaurer l'intégralité de la progression, de l'équipement et des cristaux accumulés par l'utilisateur lors du rechargement de la partie.

Tutoriel

Le niveau d'introduction, situé dans l'environnement du volcan (Mount Emberfall), représente le premier point de contact entre le joueur et notre moteur de jeu. Mon objectif n'était pas de concevoir une simple succession de panneaux textuels figés, mais de développer un système d'apprentissage (onboarding) totalement dynamique, immersif et adaptatif. Ce système devait analyser en temps réel le comportement du joueur pour valider ses actions (se déplacer, sauter, attaquer, ouvrir l'inventaire) et réagir de manière intelligente au périphérique d'entrée utilisé. En effet, pour garantir une expérience utilisateur (UX) optimale, le jeu devait être capable de détecter si l'utilisateur jouait au clavier,

avec une manette Xbox, ou avec une manette PlayStation, afin de modifier instantanément l'interface visuelle et d'afficher les icônes de touches correspondantes à l'écran.

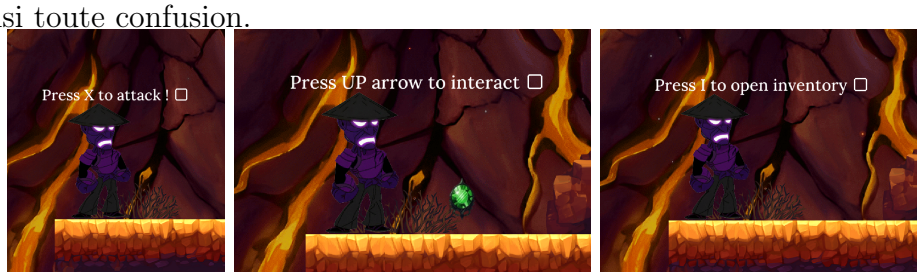
L'architecture du tutoriel repose sur un paradigme de programmation événementielle couplé à une machine d'états d'interface. Pour éviter d'alourdir la boucle physique de mise à jour du niveau, j'ai isolé la détection de l'apprentissage dans une fonction dédiée à l'interception des événements bruts. Dès qu'une action requise est effectuée (comme l'inclinaison de l'axe d'un joystick ou la pression de la touche d'attaque paramétrée), un moniteur d'état booléen est définitivement basculé sur vrai.

Pour l'affichage adaptatif, j'ai implémenté un algorithme de détection matérielle au moment de l'initialisation du niveau. L'algorithme interroge le pilote matériel via le module de joystick pour récupérer la chaîne de caractères identifiant la manette. Par une analyse syntaxique cherchant des mots-clés spécifiques (comme "ps4", "dualsense" ou "playstation"), le moteur détermine la famille du contrôleur et charge un dictionnaire d'interface spécifique.

Le véritable défi technique se trouvait dans le moteur de rendu du texte hybride. Les instructions à l'écran mélangent du texte standard et des images (les icônes des boutons). J'ai conçu un algorithme d'assemblage visuel itératif : le système lit des tableaux de données contenant des séquences d'instructions (par exemple : "Press", "cross", "to jump"). Pour chaque élément, il vérifie dynamiquement si le mot correspond à une ressource graphique chargée en mémoire. Si c'est le cas, il dessine la surface de l'image, calcule mathématiquement sa largeur en pixels, et décale le curseur de rendu horizontalement avant de dessiner la suite du texte. Enfin, un algorithme géométrique génère une case à cocher à côté de l'instruction et y trace dynamiquement une validation verte dès que le moniteur d'état de l'action passe à vrai.

Lors des premiers tests utilisateurs, un problème majeur d'ergonomie a émergé concernant le séquençage des actions. Initialement, l'interface demandait au joueur d'attaquer un monstre d'entraînement (le Golem du tutoriel) situé à une certaine coordonnée. Une fois le monstre vaincu, l'instruction suivante, invitant le joueur à interagir pour ramasser le butin et ouvrir son inventaire, apparaissait à une coordonnée fixée statiquement beaucoup plus loin dans le niveau. Conséquence : les joueurs tuaient le monstre, mais ne voyaient pas la suite des instructions apparues hors du champ de la caméra, ce qui bloquait totalement leur progression.

Pour corriger ce défaut critique sans figer l'action, j'ai dû repenser la logique de rendu pour qu'elle devienne contextuelle plutôt que strictement spatiale. J'ai injecté une dépendance directe entre l'algorithme d'affichage et l'état en mémoire du monstre ciblé. Désormais, la boucle de rendu scrute en temps réel le statut vital de l'instance du Golem. Tant qu'il est en vie, les instructions d'interaction et d'inventaire sont volontairement ignorées par le processeur graphique. À la fraction de seconde où l'entité est déclarée morte, le moteur masque l'instruction d'attaque et injecte les instructions suivantes exactement aux mêmes coordonnées spatiales. Ce chaînage conditionnel garantit que l'information apparaît de manière fluide sous les yeux du joueur, à l'endroit précis de son action, éliminant ainsi toute confusion.



La construction de ce niveau tutoriel a nécessité une synchronisation avec quasiment tous les sous-systèmes du projet. En premier lieu, la détection des touches clavier a dû être interfacée dynamiquement avec le gestionnaire global des paramètres développé par l'équipe, afin que le tutoriel affiche bien les touches personnalisées du joueur et non des valeurs codées en dur. De plus, mon algorithme d'affichage d'interface a été étroitement lié au gestionnaire d'assets (AssetManager). Plutôt que de charger les dizaines d'icônes de manettes de manière globale, ce qui aurait alourdi la mémoire RAM, le niveau tutoriel exécute une fonction d'injection qui demande au gestionnaire de chercher, redimensionner et allouer dynamiquement uniquement le sous-dossier d'icônes ("ps" ou "xbox") correspondant au périphérique détecté.

Enfin, la conclusion logique du tutoriel nécessitait la mort du Golem d'entraînement. J'ai donc connecté les mécaniques de mort de ma classe Ennemi au système d'Inventaire développé par mes camarades. L'enchaînement est total : l'affichage réagit à la mort gérée par la physique, la mort déclenche le système de butin, et le ramassage de ce butin permet au joueur de valider la dernière étape de mon tutoriel en ouvrant l'interface d'inventaire, le préparant ainsi à affronter le premier vrai Boss du jeu, Magmaroth, qui l'attend dans la zone suivante.

Système de sauvegarde

Dans un jeu d'action et de plateforme axé sur la progression et l'exploration comme Reach of Vespers, la gestion de la persistance des données et le droit à l'erreur du joueur sont deux piliers fondamentaux de l'expérience utilisateur. Mon objectif dans cette section du projet était double : concevoir un système de sauvegarde robuste capable de figer l'intégralité de l'état du jeu à tout moment, et implémenter des points de contrôle fonctionnels faisant office d'ancres de réapparition dynamiques au sein des niveaux.

La sauvegarde ne devait pas simplement retenir une coordonnée brute, mais encapsuler l'état d'évolution complet de l'avatar, incluant sa santé maximale, ses compétences débloquées, ses finances, ainsi que la structure complexe de son inventaire. Concernant les points de contrôle, l'enjeu était de dépasser le simple déclencheur invisible en créant des entités interactives gratifiantes. J'ai pris la responsabilité de concevoir et d'animer entièrement ces objets pour offrir un retour visuel clair et esthétique au joueur, matérialisant son âme (sa progression) par une transition d'animation soignée et un effet de lévitation calculé mathématiquement, renforçant l'identité mystique de notre univers.

Pour le système de persistance, j'ai opté pour une architecture basée sur la sérialisation de données au format JSON, une solution standardisée offrant un excellent compromis entre lisibilité et légèreté. Le moteur de sauvegarde fonctionne de manière décentralisée par un mécanisme d'encapsulation ascendante. Chaque composant critique du jeu expose une interface de sérialisation dédiée. Lors du déclenchement d'une sauvegarde, le gestionnaire principal centralise ces dictionnaires pour générer un fichier unique écrit de manière asynchrone sur le disque de l'utilisateur. Pour éviter les conflits de droits d'écriture sous l'environnement Windows et anticiper les contraintes de déploiement, j'ai configuré le système pour qu'il cible automatiquement le répertoire système des données d'applications locales. La phase de rechargement effectue le chemin inverse : le fichier est parsé, l'intégrité de la structure est vérifiée, puis les données sont injectées dans les constructeurs des objets pour restaurer fidèlement l'état initial.

Du côté de l'interface utilisateur, la sélection des fichiers de progression repose sur un algorithme de tri temporel dynamique. Au lieu d'afficher les sauvegardes de manière arbitraire, le menu interroge les métadonnées du système de fichiers de l'appareil. Le script

récupère le marqueur temporel de dernière modification de chaque fichier, puis applique un algorithme de tri inversé combiné à un tri alphanumérique de sécurité. Cela garantit que la partie la plus récente soit perpétuellement présentée en tête de liste, optimisant l'ergonomie générale dès le lancement de l'application.

Concernant les points de contrôle, j'ai développé une classe dédiée héritant des propriétés physiques de notre moteur de sprites. L'entité est régie par une machine à trois états distincts : l'inactivité par défaut, la phase de transition d'activation, et l'état d'activité permanent. Pour donner vie à cet objet sans surcharger la mémoire vidéo avec des boucles d'images massives, j'ai combiné deux techniques d'animation. La première est une animation par images clés pour la phase de transition. Lorsque le joueur entre en collision avec la zone d'effet, l'algorithme bloque les interactions répétées, bascule l'état visuel et déroule une séquence de frames à une cadence fixe contrôlée par le delta time du processeur, simulant l'allumage progressif de la structure. La seconde technique intervient une fois l'objet actif : pour créer l'effet de lévitation fluide de la gemme supérieure, j'ai implémenté une équation d'oscillation harmonique. En exploitant une fonction trigonométrique sinusoïdale alimentée par un compteur de temps continu, le sprite calcule à chaque frame un décalage vertical précis par rapport à son ancrage initial au sol. Ce mouvement purement mathématique simule une flottabilité parfaite à l'écran de manière extrêmement économique en ressources graphiques.

Le développement du système de chargement a été confronté à un bug critique d'accès aux fichiers lors de l'intégration des menus de sélection. Lors des tests sur différentes machines, le jeu s'effondrait instantanément avec une erreur de fichier introuvable dès que le joueur cliquait sur le bouton de chargement, alors même que les fichiers de sauvegarde apparaissaient visuellement à l'écran dans la liste des choix disponibles. En analysant la pile d'exécution et les appels système, j'ai découvert une faille dans la portée des chemins d'accès de l'algorithme de tri temporel. La fonction chargée de lister les fichiers lisait correctement le répertoire sécurisé de l'application et renvoyait une liste contenant uniquement les noms des fichiers. Cependant, l'étape suivante, chargée de calculer le timestamp de modification via le système d'exploitation, tentait de lire ces noms de fichiers directement depuis le répertoire de travail courant de l'exécutable, c'est-à-dire à la racine du projet. Si l'environnement de développement ou le raccourci de lancement modifiait le dossier d'exécution, Python cherchait la sauvegarde au mauvais endroit et déclenchait un crash fatal.

Pour résoudre définitivement cette vulnérabilité, j'ai restructuré l'isolation des che-

mins au sein de notre gestionnaire de fichiers. J'ai implémenté une fonction de résolution d'adresse absolue qui sert de passerelle unique pour toutes les opérations d'entrée et de sortie. Désormais, l'algorithme de tri n'utilise plus jamais de chaînes de caractères relatives ; il convertit chaque nom de fichier trouvé en un chemin absolu normalisé et vérifié avant d'interroger le système d'exploitation. Cette abstraction a totalement immunisé le jeu contre les variations de répertoires d'exécution et a stabilisé le menu de chargement sur toutes les plateformes.

La mise en place de la persistance et des points de contrôle a nécessité des connexions profondes avec l'intégralité du code produit par mes collègues. Au niveau du Level Design, les points de contrôle ne sont pas positionnés au hasard : leur présence est détectée dynamiquement lors du chargement des cartes de tuiles. Le script lit le calque d'objets conçu sous l'éditeur de niveaux, extrait les propriétés personnalisées de la tuile, puis instancie automatiquement la classe d'animation aux coordonnées géométriques exactes prévues par notre concepteur de niveaux.

L'intégration avec le contrôleur du joueur est fusionnelle. Lorsqu'une collision est validée par le point de contrôle, l'entité communique directement avec la classe du personnage pour écraser ses coordonnées de réapparition de sécurité par ses propres coordonnées d'ancrage midbottom. De plus, lors de l'activation, le point de contrôle appelle le gestionnaire de sauvegarde pour figer instantanément l'état global du jeu, servant ainsi de "safe zone" physique et logique.

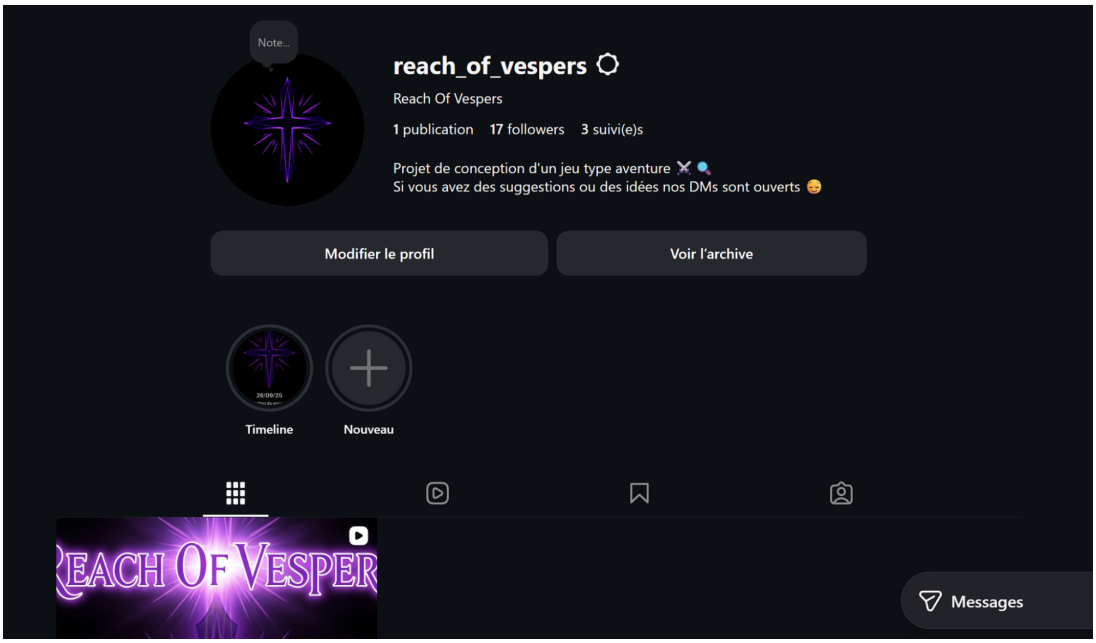
Enfin, la sérialisation JSON s'interface directement avec l'inventaire et le système d'équipement codés par mes partenaires. La méthode de sauvegarde de mon contrôleur joueur descend de manière récursive dans l'objet inventaire, convertit la grille d'objets, le nombre de fragments d'épée collectés et le type de blindage équipé en chaînes de données sémantiques, garantissant que lors du rechargement de la partie, le joueur retrouve son avatar doté exactement de la même puissance offensive et financière qu'au moment de son interaction avec le point de contrôle.

Marketing

Le développement technique de Reach of Vespers ne s'arrêtait pas à la production de son code source ; il s'agissait également d'assurer sa distribution et de fédérer une communauté de joueurs autour de notre univers. Ma mission s'est donc étendue à la conception d'une infrastructure d'acquisition d'utilisateurs, articulée autour d'un site web vitrine et d'une présence active sur les réseaux sociaux. L'enjeu principal était de concevoir cette visibilité non pas comme un simple portail statique, mais comme un véritable entonnoir de conversion optimisé techniquement pour capter le trafic, capter l'attention du public, et rediriger ces flux vers le téléchargement de notre exécutable. Il fallait prouver que notre projet d'ingénierie pouvait non seulement fonctionner, mais aussi rencontrer son public cible dans un environnement numérique ultra-concurrentiel.

Pour maximiser notre portée organique, j'ai adopté une approche d'ingénierie appliquée au marketing digital. Sur notre site web, j'ai implémenté un ensemble de techniques de référencement naturel (SEO – Search Engine Optimization). L'architecture du site a été pensée autour d'un balisage HTML sémantique strict, facilitant le travail d'analyse des robots d'indexation des moteurs de recherche. J'ai intégré des balises méta dynamiques, structuré le maillage interne autour du lexique spécifique du jeu vidéo indépendant, et mis en place des requêtes HTTP optimisées pour garantir une exploration rapide par les algorithmes de référencement.

En parallèle, pour générer du trafic d'acquisition, j'ai orchestré la création d'un compte Instagram. L'algorithme de recommandation de ce réseau privilégiant la rétention et l'interaction, j'ai adapté nos exports de jeu pour exploiter les métriques d'engagement, utilisant des formats vidéo verticaux à forte rétention pour maximiser la distribution organique de notre contenu avant de rediriger les spectateurs vers notre domaine principal.



Cette stratégie de visibilité et d'acquisition s'est profondément imbriquée avec le travail de production de l'ensemble du groupe. L'alimentation continue du compte Instagram et du site web nécessitait une synchronisation constante avec le pôle artistique et les développeurs responsables des mécaniques de gameplay. Je devais extraire des séquences vidéo pertinentes, des éléments d'interface ou des sprites de boss sans perturber leur flux de travail sur le moteur Pygame, exigeant une gestion de version logicielle rigoureuse.

3.4.5 Tristan Lempereur : Inventaire, Shop et Sound Design

En tant que responsable de l'inventaire, du shop et du sound design de Reach of Vespers, ma contribution s'articule autour de deux axes complémentaires : d'une part, la conception d'interfaces joueur permettant de gérer, utiliser et acheter des objets ; d'autre part, l'intégration sonore qui donne vie et profondeur à l'univers du jeu. Ces deux piliers, bien que distincts techniquement, partagent un objectif commun : enrichir l'expérience utilisateur et renforcer l'immersion dans le monde de Vespers.

Système d'Inventaire

A. Présentation de la fonctionnalité

L'inventaire est le point de convergence de toute la progression du joueur dans Reach of Vespers. Sans un système d'inventaire fiable et agréable à utiliser, les gemmes ramassées sur les ennemis, les armures récupérées dans les coffres et les fragments d'épée dispersés dans les niveaux n'auraient aucune signification ludique concrète. Mon objectif était donc de créer une interface complète, intuitive et visuellement cohérente avec la direction artistique sombre et mystique du jeu.

L'inventaire se présente sous la forme d'une grille de 8 colonnes × 3 lignes, soit 24 emplacements disponibles, superposée en overlay semi-transparent au-dessus de l'action en cours. Elle peut être ouverte à tout moment en appuyant sur la touche I ou le bouton B de la manette, sans interrompre le jeu (sauf si la logique de pause est activée par ailleurs). Le joueur peut y naviguer au clavier via les flèches directionnelles, à la souris via un clic, ou avec le joystick de la manette.

Chaque case de la grille peut contenir un objet, affiché avec son icône spritée, son nom et si la quantité dépasse 1, un badge numérique en bas à droite de la case. Lorsqu'un objet est sélectionné, un tooltip en bas de la fenêtre affiche son nom complet et sa description. Un menu contextuel d'action (Use/Equip+Back) apparaît à côté de la case pour permettre d'interagir avec l'objet.

Les objets gérés par l'inventaire couvrent plusieurs catégories :

— Les gemmes consommables (Healing Gem, Damage Gem, Speed Gem, Jump Gem, Dash Gem) qui confèrent des bonus temporaires ou soignent le joueur ;

— Les armures équipables (Crimson Warden, Wraithguard Armor, Voidborn Shell, Crystal Bloom Plate) qui modifient la valeur d'armure du personnage ;

— Les fragments d'épée permanents (Sword Fragment I, II, III) qui augmentent les dégâts d'attaque de façon définitive ;

— La Vespers Edge, l'épée légendaire finale forgée à partir des trois fragments.

B. Choix techniques et architecture du code

Structure de données : la grille 2D

L'inventaire est modélisé par une liste de listes Python, indexée par colonne puis par ligne. Chaque case contient soit `None` (case vide), soit une instance de la classe `Item`. Ce choix d'un tableau bidimensionnel (colonnes × lignes) est motivé par la nature visuelle de la grille : x représente la position horizontale et y la position verticale, ce qui facilite la navigation au clavier (dx, dy) et le mapping direct coordonnées à pixel à l'écran.

La classe Item :

Chaque objet est encapsulé dans une instance de la classe `Item`. Cette classe centralise les métadonnées (nom, quantité, couleur de fallback, description) et expose deux méthodes de sérialisation pour la compatibilité avec le système de sauvegarde JSON développé par Clément :

```
def __init__(self, name: str, amount: int = 1):
    self.name = name
    self.amount = amount
    self.color = Item.colors.get(name, (160, 160, 160))
    self.description = Item.descriptions.get(name, "Un␣objet␣
       mysterieux.")

def to_dict(self) -> dict:
    return {"name": self.name, "amount": self.amount}

@staticmethod
def from_dict(data: dict) -> "Item":
    return Item(data["name"], data["amount"])
```

Algorithme d'ajout avec stacking automatique

L'une des décisions algorithmiques les plus importantes concerne la méthode `add_item()`. Plutôt que de chercher directement une case vide, elle effectue d'abord un balayage complet de la grille pour détecter si un stack du même objet existe déjà. Si c'est le cas, elle incrémente la quantité existante. Ce n'est que si aucun stack n'est trouvé qu'elle cherche une case vide :

```
def add_item(self, item: Item) -> bool:
    for x in range(GRID_COLS):
        for y in range(GRID_ROWS):
            slot = self.grid[x][y]
            if slot and slot.name == item.name:
                slot.amount += item.amount
                return True

    for x in range(GRID_COLS):
        for y in range(GRID_ROWS):
            if self.grid[x][y] is None:
                self.grid[x][y] = item
                return True
    return False
```

Rendu : mise en cache du fond de panneau

Le fond de l'inventaire (panneau semi-transparent arrondi + bordure violette) est une Surface PyGame coûteuse à reconstruire à chaque frame. J'ai implémenté un système de mise en cache simple : le panneau n'est reconstruit que si la résolution de l'écran change (ce qui est rarissime en jeu). Un tuple (screen_w, screen_h) est utilisé comme clé de cache :

```
def _build_panel(self, screen_w: int, screen_h: int) -> pygame.
   Surface:
    """Construit␣la␣surface␣de␣fond␣du␣panneau␣(mise␣en␣cache)."""
    surf = pygame.Surface((INV_WIDTH, INV_HEIGHT), pygame.SRCALPHA)
    #fond arrondi
    pygame.draw.rect(surf, PANEL_COLOR, (0, 0, INV_WIDTH,
       INV_HEIGHT), border_radius=14)
    pygame.draw.rect(surf, PANEL_BORDER, (0, 0, INV_WIDTH,
       INV_HEIGHT), 3, border_radius=14)
    self._panel_surf = surf
```

```
        self._panel_key = (screen_w, screen_h)
        return surf
```

Système de buffs temporaires via l'inventaire

Lorsque le joueur utilise une gemme (Damage Gem, Speed Gem, Jump Gem, Dash Gem), la méthode `use_selected_item()` appelle la méthode `add_temporary_buff()` de la classe Joueur développée par Clément. Cette interface commune permet de découpler proprement l'inventaire du contrôleur joueur : l'inventaire ne modifie pas directement les stats, il délègue au joueur en passant le nom de la statistique, la valeur du bonus et la durée en frames :

```
elif item.name == "Jump Gem":
    player.add_temporary_buff("jump_power", 8, 5400)
    self.remove_item(item.name, 1)
    self.action_menu_open = False
    self.action_item = None
    return "Jump boosted for 90s"
```

C. Intégration avec le reste du projet

L'inventaire n'est pas un module isolé : il se connecte à plusieurs systèmes critiques du projet.

- **Connexion avec le système de sauvegarde (Clément) :** la classe `Inventory` expose les méthodes `to_dict()` et `load_from_dict()` qui permettent à la classe Joueur de Clément de sérialiser et restaurer l'état complet de la grille dans un fichier JSON. Lors du rechargement d'une partie, la grille est reconstituée fidèlement, item par item, avec les quantités exactes sauvegardées.
- **Connexion avec le système de butin :** lorsqu'un ennemi est vaincu ou qu'un coffre est ouvert, le code de Thomas appelle directement `inventory.add_item(item)` sur l'inventaire du joueur. L'inventaire est donc le réceptacle naturel de tout ce que le jeu distribue au joueur.

— **Connexion avec la classe Joueur :** `use_selected_item()` reçoit l'objet `player` en paramètre et interagit directement avec ses attributs (`health`, `armor`, `equipped_armor`) et ses méthodes (`add_temporary_buff`). Cette interface publique claire sépare les responsabilités : l'inventaire gère la possession des objets, le joueur gère leurs effets.

— **Connexion avec le Shop :** lorsqu'un achat est validé dans le shop, la transaction aboutit par un appel à `inventory.add_item()`, complétant ainsi la boucle économique du jeu : les améthystes gagnées sur les ennemis sont dépensées dans le shop, et les articles achetés atterrissent dans l'inventaire.

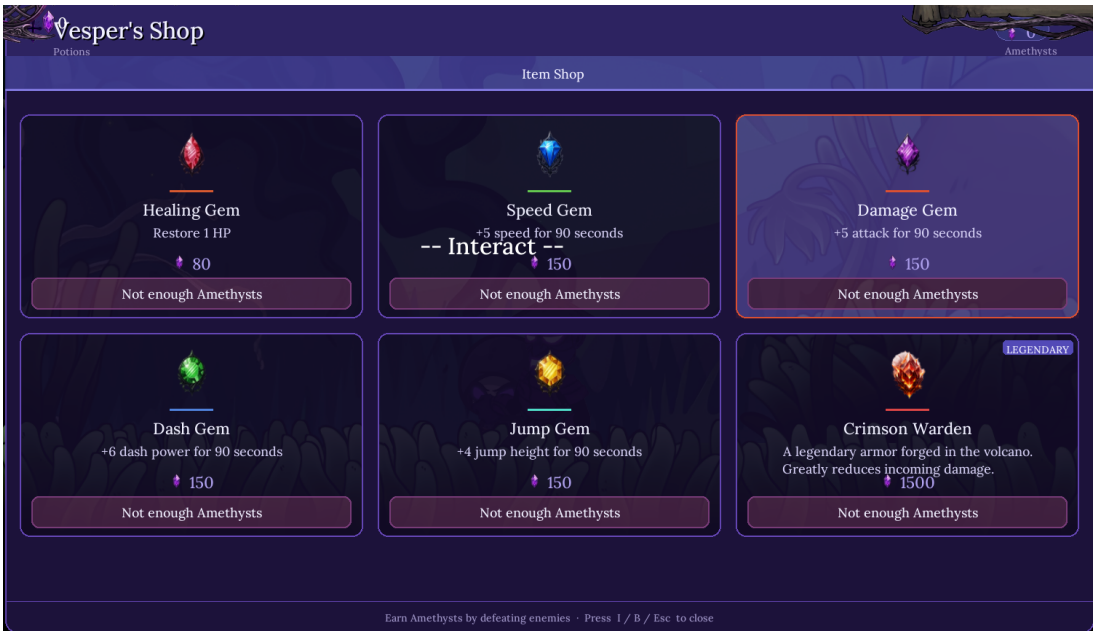
Le Shop (Vesper's Shop)

A. Présentation de la fonctionnalité

Le Shop, ou "Vesper's Shop", est l'interface de commerce du jeu. Accessible depuis certains points stratégiques des niveaux, il permet au joueur de dépenser les améthystes accumulées en combattant les ennemis pour acquérir des objets utiles à sa progression. Le shop constitue l'un des piliers de la boucle économique du jeu : tuer → récolter → acheter → progresser.

Visuellement, le shop occupe 78% de la largeur et 80% de la hauteur de l'écran, centré sur la fenêtre de jeu. Il affiche une grille de cartes produits (3 colonnes) avec pour chaque article : une icône, un nom, une description courte, un prix en améthystes et un bouton d'achat. L'interface s'anime à l'ouverture avec un effet de zoom et de fondu (scale + alpha), et affiche un compteur de la monnaie actuelle du joueur dans l'en-tête.

Le catalogue comprend 6 articles : cinq gemmes consommables (Healing Gem à 80 améthystes, Speed/Damage/Dash/Jump Gem à 150 chacune) et une armure légendaire (Crimson Warden à 1500 améthystes). Les prix ont été calibrés en concertation avec l'équipe pour offrir un équilibre entre accessibilité et sentiment de récompense.



B. Choix techniques et algorithmes

Scaleur relatif à la résolution

Le premier défi du shop était de le rendre indépendant de la résolution d'affichage. Inspiré par l'approche de Jules pour le reste de l'interface, j'ai implémenté une méthode utilitaire `_s()` qui scale toute valeur dimensionnelle proportionnellement à la hauteur de l'écran de

référence (1080p). Cela garantit que les boutons, marges et textes s'affichent de manière cohérente quelle que soit la configuration matérielle :

```
def _s(self, v):
    return max(1, int(v * self.screen_h / 1080))
```

Animation d'ouverture :
L'effet d'ouverture du shop (zoom + fondu) repose sur un paramètre d'animation `anim_open` qui progresse de 0.0 à 1.0 à chaque appel de `update()`. La fonction `_lerp()` (linear interpolation) est utilisée pour animer le scroll de manière fluide, et le scale est calculé à la volée lors du rendu :

```
def _lerp(self, a, b, t):
    return a + (b - a) * t

def update(self, dt):
    if not self.is_open:
        return
    self.anim_open = min(1.0, self.anim_open + dt * 6)
    self.scroll_y = self._lerp(self.scroll_y, self.scroll_target,
       min(1.0, dt * 12))

    if self.toast_timer > 0:
        self.toast_timer -= dt

    # Dans la fonction de dessin :
    # scale = self._lerp(0.85, 1.0, self.anim_open)
    # alpha = int(255 * self.anim_open)
```

Gestion universelle des inputs (clavier, souris, manette)
Le shop doit être pleinement jouable aussi bien à la souris qu'au clavier et à la manette. La méthode `handle_event()` centralise tous les types d'événements PyGame et les redirige vers les mêmes fonctions métier (`_try_buy_item`, `move_selection`, `close`). Pour la manette, j'ai géré les deux types d'axes analogiques courants et les touches numériques, avec un timer anti-rebond pour le joystick afin d'éviter les défilements involontaires :

```
if event.type == pygame.JOYAXISMOTION:
    if event.axis in (0, 1):
        now = pygame.time.get_ticks()
        if abs(event.value) > 0.5 and now - self.stick_timer > 200:
            self.stick_timer = now
            self.hovered_item = None
            if event.axis == 0:
                if event.value > 0.5: self.move_selection(1)
                else: self.move_selection(-1)
            elif event.axis == 1:
                if event.value > 0.5: self.move_selection(3)
                else: self.move_selection(-3)
            return True
```

Algorithme d'achat et système de toast
La logique d'achat dans `_try_buy_item()` effectue deux vérifications séquentielles : d'abord la solvabilité du joueur, puis pour l'armure Crimson Warden spécifiquement, la détection d'un doublon dans l'inventaire (pour éviter d'acheter deux fois le même article

non consommable). Si l'achat est valide, la transaction est enregistrée dans une file `_purchases`, qui sera lue par la boucle principale du jeu pour créditer le joueur et mettre l'objet dans son inventaire. Cette architecture en file de transactions découple le shop de la logique du joueur.

C. Problèmes rencontrés et solutions apportées

Problème 1 : les images d'armure introuvables au déploiement

Lors des tests de déploiement avec l'exécutable compilé par Jules via PyInstaller, les icônes d'armure n'apparaissaient plus dans le shop — remplacées par des rectangles de couleur unie. Le problème venait du fait que les armures n'étaient pas toujours préchargées dans l'objet `assets` fourni au constructeur du Shop, et que les chemins relatifs vers les fichiers PNG différaient selon que le jeu tournait en mode développement (depuis les sources) ou en exécutable compilé.

La solution a été de créer une méthode `_load_armor_images()` qui tente plusieurs chemins de fichiers alternatifs pour chaque armure, et stocke les images chargées soit dans `self.assets` (si disponible), soit dans un dictionnaire de fallback `self._local_assets`. Cette résolution multi-chemin rend le shop robuste aux différences d'arborescence :

```
armor_keys = {
    "item_armor1": [
        "assets/game/items/armor1.png", "assets/game/items/armor_1.
           png",
        "assets/game/hud/items/armor1.png", "assets/game/potions/
           armor1.png"
    ],
    "item_armor2": [
        "assets/game/items/armor2.png", "assets/game/items/armor_2.
           png",
        "assets/game/hud/items/armor2.png", "assets/game/potions/
           armor2.png"
    ],
    "item_armor3": [
        "assets/game/items/armor3.png", "assets/game/items/armor_3.
           png",
        "assets/game/hud/items/armor3.png", "assets/game/potions/
           armor3.png"
    ],
    "item_armor4": [
        "assets/game/items/armor4.png", "assets/game/items/armor_4.
           png",
        "assets/game/hud/items/armor4.png", "assets/game/potions/
           armor4.png"
    ]
}

if not hasattr(self, "_local_assets"):
    self._local_assets = {}

for key, paths in armor_keys.items():
    if self.assets is not None and getattr(self.assets, key, None)
       is not None:
        continue
```

```
    for path in paths:
        if os.path.exists(path):
            try:
                img = pygame.image.load(path).convert_alpha()
                if self.assets is not None:
                    setattr(self.assets, key, img)
                else:
                    self._local_assets[key] = img
                break
            except:
                pass
```

Problème 2 : les rects de clic non réinitialisés entre les frames
Un bug intermittent faisait que certains clics sur les boutons "Buy" ne déclenchaient aucune action, tandis que d'autres clics dans des zones vides achetaient accidentellement un article. L'origine du problème : la liste `_item_buy_rects` qui mappe les boutons à leurs Rect de collision n'était pas réinitialisée avant chaque frame de dessin. Des rects d'une frame précédente (potentiellement avec d'autres dimensions si le scroll avait changé) persistaient dans la liste.

La correction a été aussi simple qu'efficace : réinitialiser `_item_buy_rects` au début de chaque appel à `draw()`. Les rects sont recalculés et réenregistrés à chaque rendu, garantissant une correspondance parfaite entre ce qui est visible à l'écran et les zones de clic actives :

```
def draw(self, screen, player_crystals):
    if not self.is_open:
        return
    self._item_buy_rects = []
    # Reste du rendu ...
    # self._draw_items_tab(surf, player_crystals)
```

D. Intégration avec le reste du projet

Le shop communique avec le reste du jeu via le pattern de la file de transactions. La boucle principale (`main_game.py`) appelle `shop.pop_purchase()` à chaque frame pour récupérer d'éventuelles transactions. Si une transaction est présente, le jeu la traite : il débite les améthystes du joueur et appelle `player.inventory.add_item()` pour créditer l'objet.

Cette architecture en file découple proprement le shop de tous les autres modules. Le shop ne connaît pas la classe Joueur directement (sauf pour la vérification de doublon d'armure) — il ne fait que produire des transactions. C'est la boucle principale qui orchestre la mise à jour des données du joueur. Cela a facilité les tests unitaires du shop et évité des dépendances circulaires entre modules.

Sound Design

A. Présentation de la fonctionnalité

Le sound design de Reach of Vespers recouvre l'intégralité de l'habillage sonore du jeu : musiques d'ambiance par biome, effets sonores des actions du joueur, sons de combat, feedbacks audio des interfaces (inventaire, shop) et sons des ennemis. Son rôle est fondamental pour créer l'immersion dans un Metroidvania : l'atmosphère d'un marais toxique ne se construit pas uniquement avec des visuels, elle se ressent par ses sons.

Tous les assets audio ont été sélectionnés sur Pixabay, une plateforme proposant des sons libres de droits. Ce choix a été dicté par les contraintes légales du projet (pas de musique sous copyright dans un projet académique distribué publiquement) et par la richesse de la bibliothèque disponible. Pour chaque biome, j'ai constitué une banque de sons thématiquement cohérente :

- **Mount Emberfall (volcan) :** percussions lourdes, grondements graves, crépitements de lave ;
- **Toxic Marsh (marais) :** ambiances organiques, bourdonnements, cordes dissonantes ;
- **Mushroom Caverns (cavernes fongiques) :** sons cristallins, réverbérations profondes, atmosphère mystérieuse ;
- **Ruined City (cité dévastée) :** vents urbains, cuivres sombres, tension dramatique.

En plus des musiques d'ambiance, intégration des effets sonores pour toutes les actions significatives : attaque à l'épée, dash, saut, atterrissage, prise de dégâts, mort du joueur, ouverture de l'inventaire et du shop, achat d'article, activation d'un checkpoint, et collecte d'améthystes.

B. Choix techniques : gestion audio dans PyGame

Architecture du mixer PyGame

Pour garantir une expérience sonore fluide, j'ai utilisé l'outil `pygame.mixer`, qui permet de jouer plusieurs sons en même temps. Afin d'éviter tout conflit ou saturation, j'ai séparé les sons en deux catégories distinctes :

- **La musique d'ambiance :** Elle est gérée par un canal dédié, ce qui permet de diffuser des morceaux longs en continu sans interruption.
- **Les effets sonores (SFX) :** Ces sons courts (bruits de pas, combats, ouvertures de coffres) utilisent des canaux séparés.

Cette séparation technique est essentielle, car elle permet au menu des options (conçu par Jules) de contrôler le volume de la musique et celui des effets sonores de manière totalement indépendante. Ainsi, le joueur peut ajuster son expérience sonore à sa convenance sans que le volume global ne soit affecté.

Synchronisation sons et animations (coordination avec Clément)

L'un des plus grands défis en sound design a été de faire en sorte que les bruitages correspondent parfaitement à l'action à l'écran. Un décalage, même infime, entre le moment où le personnage frappe et le bruit de l'épée, casse immédiatement l'immersion du joueur.

Le fonctionnement technique : Nous avons intégré des points de contrôle directement dans le comportement des ennemis et du joueur. Au lieu de jouer un son aléatoirement, le programme surveille en temps réel le défilement des images de l'animation. Dès que le compteur atteint la frame précise que nous avons définie comme l'instant de l'action, le son est déclenché. Cela garantit une précision parfaite, où chaque mouvement est accompagné instantanément par son effet sonore dédié. Cette méthode assure que le retour auditif est toujours en totale adéquation avec le retour visuel, renforçant ainsi la sensation de contrôle et de réalisme pour le joueur.

C. Problèmes rencontrés et solutions apportées

Problème : saturation des canaux audio

Lors des premiers tests, nous avons rencontré un souci de saturation : dans les zones où il y avait beaucoup d'ennemis, certains sons se coupaient brusquement. Le problème venait du fait que PyGame ne gérait que 8 canaux audio simultanément, ce qui devenait insuffisant dès que l'action s'intensifiait.

Pour régler cela, j'ai simplement augmenté la capacité du système à 16 canaux, ce qui couvre désormais largement nos besoins. J'en ai profité pour instaurer une règle de priorité : les sons importants (comme les dégâts subis par le joueur) sont toujours diffusés en priorité, tandis que les sons secondaires (comme les bruits de pas d'ennemis éloignés) sont automatiquement ignorés s'ils ne sont pas assez proches pour être pertinents. Ce petit ajustement a suffi à rendre l'ambiance sonore propre et cohérente en toutes circonstances.

D. Intégration avec le reste du projet

Le sound design est le module qui se connecte au plus grand nombre de composants différents du projet. Par essence, l'audio réagit à tout ce qui se passe dans le jeu.

- **Avec l'interface :** le menu des paramètres expose deux sliders de volume (Music et SFX). Leur valeur est lue en temps réel par le gestionnaire audio pour ajuster le volume global des deux canaux. La sauvegarde de ces préférences dans le fichier de configuration local assure leur persistance entre les sessions.
- **Avec le Level Design :** chaque niveau dispose d'un tag de biome. Au chargement d'un niveau, le gestionnaire audio lit ce tag et lance la piste musicale correspondante en fondu enchaîné. Les banques de sons de pas et d'ambiance sont également sélectionnées en fonction de ce tag.
- **Avec l'inventaire et le shop :** l'ouverture et la fermeture de ces interfaces déclenchent des sons d'interface discrets. L'achat d'un article dans le shop, la validation d'une utilisation dans l'inventaire et la réception d'un message d'erreur ("Not enough Amethysts") sont chacun associés à un son spécifique, renforçant le feedback utilisateur.

Chapitre 4

Récit de la Réalisation

Le développement de Reach of Vespers s'est étalé sur environ huit mois. Passer du stade de la conception à un produit final jouable et complet a été une aventure riche en apprentissages, jalonnée de réussites motivantes mais aussi de défis techniques et organisationnels parfois intenses.

4.1 Nos réussites et nos joies

La plus grande satisfaction de notre équipe, le Studio J-1, a été de voir notre univers prendre vie. L'intégration des premiers croquis de Soren et des décors générés (assets) dans la fenêtre PyGame a constitué un véritable moteur pour notre motivation.

Sur le plan technique, nous sommes particulièrement fiers d'avoir réussi à implémenter un mode multijoueur fonctionnel, une tâche dont nous avions mesuré la forte complexité dès l'élaboration de notre cahier des charges. Au-delà de la maîtrise du langage Python et du framework PyGame, l'apprentissage de nouveaux outils professionnels comme GitHub pour le versionnage de notre code, ou OpenCV pour la lecture de nos cinématiques, figurent parmi nos plus grandes réussites.

Sur le plan humain, la cohésion de notre groupe est restée intacte de bout en bout. Le système de Responsable/Suppléant (R/S) que nous avions théorisé dans notre document de spécifications a prouvé son efficacité sur le terrain : l'entraide a toujours primé sur l'individualisme, ce qui nous a permis de maintenir un environnement de travail sain et productif.

4.2 Nos peines et les défis rencontrés

Naturellement, un projet de cette envergure ne s'est pas déroulé sans heurts. Notre principale difficulté a été la gestion du temps face à la courbe d'apprentissage de certaines technologies. Le défi le plus marquant a été notre retard accumulé lors de la deuxième soutenance. À ce stade, comme en témoigne notre tableau de suivi, le mode multijoueur n'était pas opérationnel. Parallèlement, nous accusions un déficit sur la richesse du contenu global du jeu (les objets et mini-jeux accusaient également des retards). De plus, nous avons fait face à de nombreux problèmes techniques liés aux limites matérielles de PyGame, mais aussi à des conflits récurrents lors de nos fusions de code sur Git (merge conflicts), nécessitant de longues sessions de débogage en groupe pour retrouver un exécutable stable.

4.3 Analyse critique et solutions apportées

Avec le recul, nous avons analysé les causes de nos retards : ils provenaient principalement d'une sous-estimation de la complexité de l'architecture réseau au moment de la planification initiale, couplée à une parallélisation imparfaite de nos tâches.

Pour corriger le tir, nous avons mis en place un ensemble de solutions curatives et préventives :

- **Réorganisation :** Face au retard de la Soutenance 2, nous avons basculé d'un modèle de travail individuel à un modèle par "sprints" intensifs.
- **Pivot technologique :** Plutôt que de subir les ralentissements de PyGame face à nos assets HD, nous avons pris la décision proactive de migrer le projet vers Pygame-CE, optimisant ainsi les calculs d'affichage via le GPU.
- **Devoir de communication :** Afin de limiter les régressions et les conflits sur Git, nous avons créé un salon textuel "patch-note" sur notre Discord.

Bilan prospectif : Si nous devions recommencer ce projet aujourd'hui, nous déploierions plusieurs méthodes pour éviter ces écarts entre le prévisionnel et le réel. Tout d'abord, nous réaliserions des "Preuves de Concept" (PoC) techniques beaucoup plus poussées dès les premières semaines sur les mécaniques critiques (notamment la connexion LAN) afin de mieux estimer le temps de développement nécessaire. Enfin, nous multiplierions les réunions de synchronisation courtes afin de mieux coordonner l'ensemble du groupe à chaque étape clé.

Chapitre 5

Qualité du Produit et Documentation Technique

Notre objectif avec le Studio J-1 n'était pas seulement de valider des compétences en programmation, mais de livrer un produit fini, stable et professionnel. Pour ce faire, nous avons porté une attention toute particulière à l'expérience utilisateur, à la stabilité de notre exécutable et à la qualité de notre documentation.

5.1 Direction artistique et prise en main

Conformément à la vision établie dans notre cahier des spécifications techniques, nous avons veillé à maintenir une parfaite cohérence visuelle tout au long de l'expérience. L'univers sombre et mystérieux de Reach of Vespers se reflète fidèlement dans l'ensemble des facettes du produit (des menus jusqu'aux décors des combats de boss), traduisant avec exactitude nos intentions de départ.

Afin de garantir une prise en main naturelle et intuitive pour n'importe quel type de joueur, Clément a développé un tutoriel progressif. De plus, pour ne jamais laisser le joueur dans l'incompréhension, une aide contextuelle a été directement intégrée aux éléments interactifs du jeu.

5.2 Un exécutable unique et robuste

D'un point de vue logiciel, l'intégralité de notre code et de nos ressources (assets, bibliothèques, scripts) a été compilée avec succès. Toutes les fonctionnalités développées par l'équipe (le mode multijoueur, les intelligences artificielles, les mini-jeux et l'interface) sont aujourd'hui intégrées dans un seul et unique produit exécutable.

À la suite de nos nombreuses sessions de débogage et de tests croisés, nous sommes fiers de présenter une version finale où les bugs restants ont été traqués, garantissant qu'aucun bug bloquant ne subsiste pour entraver la progression du joueur.

5.3 Déploiement et Dossier d'Exploitation

Pour qu'un logiciel soit considéré comme un produit fini, son déploiement doit être irréprochable. Nous avons donc développé une procédure complète permettant d'installer le projet de manière automatisée. Cet installateur gère la mise en place du jeu et

de ses dépendances. Conformément aux exigences techniques du projet, un utilitaire de désinstallation propre à Windows est également intégré à notre installateur. Le jeu est installé dans la liste des programmes Windows et lorsqu'on souhaite le désinstaller, le même programme se lance et désinstalle le jeu et toutes ses dépendances.

En complément du logiciel, nous livrons un dossier d'exploitation complet, dactylographié, imprimé et relié, comprenant le manuel d'installation détaillé ainsi que le manuel d'utilisation.

5.4 Rigueur de la documentation technique

L'ensemble de nos choix techniques, algorithmiques et architecturaux ont été documentés. Que ce soit au sein même de notre code source via des commentaires pertinents, ou à travers ce présent rapport, tous les aspects du produit sont détaillés et nos arguments (comme le passage à Pygame-CE ou l'architecture LAN) sont justifiés.

Chapitre 6

Conclusion Générale

Le développement de Reach of Vespers s'achève sur la livraison d'un produit informatique complet, stable et fidèle à la vision que nous avions établie dans notre cahier des spécifications techniques (CDST). Au terme de ces huit mois de développement, le Studio J-1 a su transformer des concepts ambitieux en une expérience vidéoludique concrète et immersive, mêlant habilement les mécaniques du jeu de plateforme, de l'Action-RPG et du jeu de combat (Beat them all).

Bilan technique et méthodologique

Les contraintes pédagogiques imposées - l'utilisation exclusive du framework PyGame, l'intégration d'intelligences artificielles autonomes et la conception d'un mode multijoueur en réseau local - n'ont pas été perçues comme de simples obligations, mais comme de véritables moteurs de progression. Ce projet nous a confrontés aux réalités du métier d'ingénieur : nous avons appris à justifier des pivots technologiques cruciaux (comme notre migration vers Pygame-CE ou l'adoption d'OpenCV), à concevoir des exécutables uniques exempts de bugs bloquants, et à produire une documentation d'exploitation professionnelle. De plus, la maîtrise d'outils collaboratifs tels que GitHub et la mise en place de processus de communication adaptés (salons patch-note sur Discord) ont durablement structuré notre rigueur technique.

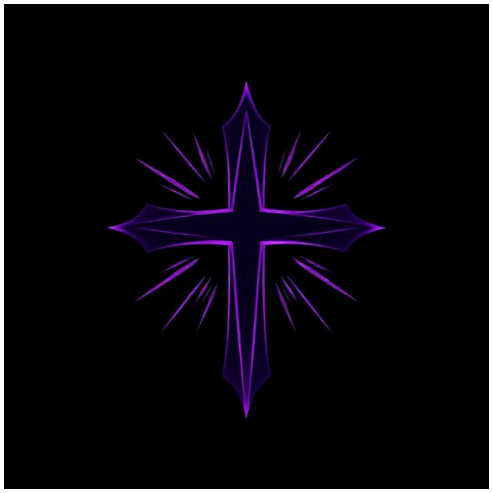
Bilan humain et dynamique de groupe

Au-delà du code source, la plus grande réussite de notre équipe réside dans la gestion de la dynamique humaine. Le système de Responsable (R) et Suppléant (S), que nous avions théorisé dès le début du projet, a prouvé toute son efficacité face à la pression et aux retards. L'entraide, la réorganisation en méthodes agiles ("sprints") et la répartition intelligente des tâches en fonction des compétences de chacun ont permis de maintenir un niveau de motivation constant. Aujourd'hui, chaque membre du groupe - Jules, Thomas, Thibault, Clément et Tristan - possède une vision globale et transversale de l'architecture du jeu.

En livrant cet exécutable accompagné de son installateur et de son manuel d'exploitation, nous ne clôturons pas simplement notre première année à l'EPITA : nous concrétisons une première expérience professionnelle aboutie en ingénierie informatique.

Reach of Vespers est désormais prêt à dévoiler ses mystères aux joueurs.

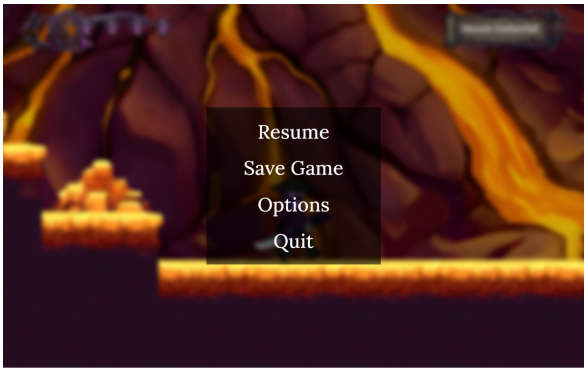
J-1 Studios



Chapitre 7

Annexe

7.1 Le HUD et le menu Pause



7.2 Les items